TAMPERE UNIVERSITY OF TECHNOLOGY

TOMI ÄIJÖ

INTEGER LINEAR PROGRAMMING BASED CODE GENERATION FOR EXPOSED DATAPATH

Master of Science Thesis

Examiners: Prof. Tapio Elomaa and D.Sc. Pekka Jääskeläinen
Examiners and topic approved in the Faculty of Computing and Electrical Engineering Council meeting
15th of January 2014

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
TOMI ÄIJÖ: Integer Linear Programming Based Code Generation for Exposed Datapath
Master of Science Thesis, 52 pages, 2 Appendix pages
December 2014
Major: Computer science
Examiners: Prof. Tapio Elomaa and D.Sc. Pekka Jääskeläinen
Keywords: processors, transport triggered architecture, instruction level parallelism, compilers, integer linear programming

As the use of embedded processors has spread throughout the society pervasively, the requirements for the processors have become much more diverse causing general purpose processors to be inefficient on many occasions. This creates the need for customized processors that are tailored for a particular use case. Transport triggered architecture is a processor architecture template that exploits the instruction level parallelism. The architecture provides the basic building blocks and means to construct custom tailored processors. Transport triggered architecture processors are statically scheduled, thus powerful instruction scheduling algorithms can bring up significant efficiency increases in terms of chip area, clock frequency, and energy consumption.

This thesis proposes an integer linear programming model for the instruction scheduling problem of the transport triggered architecture. The model describes the architecture characteristics, the particular processor resource constraints, and the operation dependencies of the scheduled program. It is possible to optimize the model for various criterion, for example to achieve as energy efficient processors as possible. This scheduling algorithm was implemented to the high-level language compiler of the TTA-based Co-design Environment, which is a toolset for designing processors using the transport triggered architecture template.

The model was tested and measured with example problems such as complex number arithmetics, and vector dot product. Such example algorithms are typically executed in embedded processors and parallelize reasonably well. The performance results were compared to the existing heuristic graph-based scheduling algorithm of the toolset compiler.

The study indicates that the integer linear programming based instruction scheduler produced significantly shorter schedules compared to the heuristic scheduler. In addition, the amount of register access in the compiled programs was generally notably less than those of the heuristic scheduler. On the other hand, the proposed scheduler used distinctly more execution time than the heuristic scheduler.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
TOMI ÄIJÖ: Kokonaislukuoptimointiin perustuva koodigenerointi näkyvän datapolun arkkitehtuureille
Diplomityö, 52 sivua, 2 liitesivua
Joulukuu 2014
Pääaine: Ohjelmistotiede
Tarkastajat: prof. Tapio Elomaa ja TkT Pekka Jääskeläinen
Avainsanat: prosessorit, siirtoliipaisuarkkitehtuuri, käskytason rinnakkaisuus, kääntäjät, kokonaislukuoptimointi

Viimeisten vuosikymmenten aikana sulautetut prosessorit ovat levinneet laajalle ja niihin ei voi olla törmäämättä jokapäiväisessä elämässä. Koska prosessoreille on mitä monimuotoisempia käyttötarkoituksia, eivät yleiskäyttöiset prosessorit sovi kaikkiin tapauksiin. Tämä on johtanut tarpeeseen suunnitella räätälöityjä prosessoreita, jotka mukautetaan tiettyä käyttötarkoitusta varten. Siirtoliipaisuarkkitehtuuri on käskytason rinnakkaisuutta hyödyntävä prosessoriarkkitehtuuri, joka tarjoaa työkalut prosessorin suunnitteluun uudelleenkäytettäviä komponentteja yhdistelemällä. Arkkitehtuuri on staattisesti skeduloitu, ja tehokkaalla käskyskeduloinnilla voidaan saada aikaan suuria tehokkuuseroja piirin koon, kellotaajuuden ja energiatehokkuuden suhteen.

Tämä työ esittää siirtoliipaisuarkkitehtuuriin suunnitellun staattisen skedulointialgoritmin, joka on esitetty matemaattisena kokonaislukuoptimointimallina. Malli kuvaa arkkitehtuurin erityispiirteet, tietyn prosessorin asettamat resurssirajoitteet ja ohjelman käskyjen sisäiset riippuvuudet sekä mahdollistaa ohjelman operaatioiden aikataulun optimoinnin tietyn kriteerin perusteella. Ohjelma voidaan esimerkiksi optimoida mahdollisimman energiatehokkaaksi. Algoritmi toteutettiin osaksi siirtoliipaisuarkkitehtuuriin suunnitellun kehitysympäristön ohjelmointikielen kääntäjää.

Skedulointialgoritmi testattiin ja sen tehokkuus arvioitiin käyttäen esimerkkiohjelmia kuten kompleksilukuaritmetiikkaa, ja vektorien pistetulon laskenta. Kyseiset algoritmit ovat tyypillisiä sulautetuissa prosessoreissa suoritettavia hyvin rinnakkaistuvia ongelmia. Algoritmia verrattiin kehitysympäristön kääntäjän olemassa olevaan heuristiseen skedulointialgoritmiin.

Mittaukset osoittavat, että kokonaislukuoptimointimalliin perustuva käskyskedulointialgoritmi tuottaa huomattavasti lyhyempiä ohjelmia verrattuna edellä mainittuun heuristiseen skedulointialgoritmiin. Lisäksi rekisterien käyttöaste oli useimissa tapauksissa huomattavasti pienempi kuin heuristisen skedulointialgoritmin. Toisaalta ehdotetun skedulointialgoritmin suoritusaika oli suuruusluokkaa isompi suhteessa heuristiseen skedulointialgoritmiin.

PREFACE

The work in this M.Sc. thesis was carried out at the Department of Pervasive Computing at Tampere University of Technology as a part of the Parallel Acceleration Project (ParallaX).

I would like to express my gratitude and appreciation to Professor Tapio Elomaa, the advisor of this thesis, for his guidance, advice, and ideas through the process. I am especially grateful to Pekka Jääskeläinen, Dr. Tech, for his invaluable guidance and encouragement. Also I thank all the colleagues in the Customized Parallel Computing group for their support. Especially Heikki Kultala, M.Sc., has been a great help in the development process.

Finally, I wish to than my family for their lasting love, encouragement and perseverance throughout my studies.

Tampere, Sep 25, 2014

TOMI ÄIJÖ

CONTENTS

SYMBOLS AND ABBREVIATIONS

ADF	Architecture Definition File
ALU	Arithmetic Logic Unit
CU	Control Unit
DDG	Data Dependence Graph
GPP	General Purpose Processor
FIR	Finite Impulse Response filter
FU	Function Unit
HDL	Hardware Description Language
II	Initiation Interval
IIR	Infinite Impulse Response filter
ILP	Instruction Level Parallelism
IR	Intermediate Representation
IU	Immediate Unit
LLVM	Low Level Virtual Machine, a compiler infrastructure
LSU	Load-Store Unit
NDTM	Nondeterministic Turing Machine
NOP	No-Operation
NP	Nondeterministic Polynomial Time
MIP	Mixed Integer Programming
RAM	Random-Access Memory
RaW	Read after Write
RF	Register File
SCIP	Solving Constraint Integer Programs, a MIP solver
SoC	System-on-a-Chip
SOS	Special Ordered Set
SSA	Static Single Assignment, a type of IR
OpenCL	Open Computing Language
TCE	TTA-based Co-design Environment
TPEF	TTA Program Exchange Format

TTA	Transport Triggered Architecture
VLIW	Very Long Instruction Word
WaR	Write after Read
WaW	Write after Write

1. INTRODUCTION

For the past few decades, the landscape of *Systems-on-a-Chip* (SoC) circuits has dramatically changed: SoC circuits have become more ubiquitous than ever. Computing appears pervasively everywhere in numerous shapes and sizes. An individual interacts with SoC circuits on many different forms, often without realizing that a sophisticated computer is involved.

As computers are found practically everywhere, the requirements for a processor design have become more diverse rendering *General Purpose Processors* (GPP) ineffective on many occasions. These requirements typically involve computation performance, energy consumption, and the physical chip size. In order to cope with the growing complexity and taxing demands, many times the processor needs to be custom tailored for a specific use case, occasionally from the scratch.

Designing a complex processor from the ground up is a highly expensive and demanding operation, and at the same time, there is a need for a fast time to market due to competition. There exist numerous prefabricated architecture templates for processor construction that alleviate the cost and time of the tailored design. *Transport Triggered Architecture* (TTA) is a processor architecture template that provides basic building blocks and extensive possibilities for custom processor composition. The leading principle of TTA is that much of the complexity is moved from the hardware to the compiler. A TTA program is statically scheduled raising the need for powerful instruction scheduling algorithms to be able to meet the design requirements.

NP-complete problems are a collection of computational problems that have efficiently verifiable solutions, but no known deterministic linear time algorithm. They are the hardest problems amongst the NP, in a sense that if one can be solved in polynomial time then so can all other problems. Finding the optimal instruction schedule for a TTA program is proven to be NP-complete problem [1].

Better scheduling algorithms do not only lead to slight incremental advance, but possibly to an order of magnitude improvement. She at al. [2] presented up to 80% effective energy saving with improvements over the register access using a relatively simple heuristic scheduler for a TTA processor. Due to the rapid spread of mobile phones, the Internet of Things, and other ultra-low power devices, the demand for more power efficient SOC circuits are higher than ever. In addition to

energy savings, improved schedules permit the pruning of the internal connections of a processor which in turn allows higher clock frequencies.

The objective of this thesis project was to create an efficient and versatile instruction scheduling algorithm for the TTA template processors. The scheduler is implemented by describing the scheduling problem as a mathematical model using integer linear programming. Furthermore, the defined model is solved using an integer linear programming solver to obtain the most favorable outcome according to a specific criterion. The defined model is tested using a few benchmark programs such as complex number arithmetics, and dot product calculation. These problems are quite commonly executed in SoC circuits, and require high performance and, simultaneously, low energy consumption.

The thesis is organized as follows. The second chapter introduces the concept of customized processors more deeply. Moreover, the TTA processors are described thoroughly and an example TTA processor is presented. In addition, *TTA-based Co-design Environment* (TCE) is introduced, which can be used to design TTA processors. The described instruction scheduler was implemented into the high-level language compiler of the TCE.

Compilers, their structure, and internal data representation are discussed in Chapter 3. Furthermore, a few common data structures in compilers are presented. Chapter 3 also talks about code generation specifically for transport triggered architecture processors. The basic stages of code generation are presented, in particular we discuss instruction scheduling which is in the focus of this thesis. Integer linear programming is introduced in Chapter 4 along with two example problems, the knapsack problem and the traveling salesman problem. These problems are expressed as integer linear programming problems. Chapter 5 describes the scheduling algorithm in terms of integer linear programming. Chapter 6 lays out the results of experiments and discusses them. Chapter 7 summarizes the main results, concludes the thesis, and provides suggestion for further research regarding the integer linear programming model for the TTA scheduling problem.

2. PROCESSORS

A processor is a programmable computing unit that executes and responds to instructions. Typically processors are composed of multiple individual components that can perform various tasks. A processor is said to be *multi-issue* if there are multiple execution units that perform concurrently, as opposed to *single-issue* processors. The multi-issue processors can achieve the concurrency either *dynamically*, that is, at the runtime, or *statically* at the compilation time.

Processors are designed and optimized for a particular task in case general purpose processors do not meet the requirements. The target application determines the resources needed to carry out the computations of specific application as efficiently as possible. Processors can be tailored by restricting and expanding the operation set, or varying the physical functional units on board. In customized processor design typical objectives are chip area, power consumption, manufacturing costs, and clock rate.

Exposed datapath architectures are processor architectures where the programmer or the compiler have direct control over the datapath [3]. These types of architectures typically result in small, low-power processors that can achieve high performance by efficient compile-time scheduling and high-level of parallelism.

2.1 Instruction Level Parallelism

Instruction Level Parallelism (ILP) is a measurement of the number of operations that can be performed simultaneously in a multi-issue processor [4]. A traditional way to parallelize the execution of operations is *pipelining*, in which the operations are divided into a sequence of dependent steps or stages. These stages can be executed concurrently so that multiple operations are partly in execution at the same time. Pipelining increases instruction throughput as multiple operations can be performed simultaneously. An example of pipelining is fetching the next instruction while the previous has not been completed yet, and similarly writing the result of a completed operation to a register or memory when the next operation has already been started. Pipelining is utilized dynamically by the processor.

Another way to exploit the possible concurrency of operations is to use of the fine grained independence of operations allowing multiple operations to be executed simultaneously. This is achieved by executing the sequential operation stream con-

currently given the processor resources such that the dependencies between the operations are satisfied. The amount of static ILP that can be achieved varies greatly depending of the application. This type of ILP needs thorough analysis of the program when it is statically utilized by the compiler at compilation time.

2.2 Very Long Instruction Word Architecture

Very Long Instruction Word (VLIW) is a processor architecture style that is designed to take advantage of static instruction level parallelism of operations [5]. Traditional dynamical approaches to improving performance in processor architectures result in increased hardware complexity, whereas the static multi-issue, by contrast, moves complexity from the hardware to the compiler. VLIW processor consists of function units and register files, and an interconnection network enabling all possible concurrent data transports between the units. In other words, VLIW architecture allows the customization of the processor by varying the components. Because of this, VLIW is not a single processor but a template that can be used to design different kinds of processors.

The VLIW processor instruction stream consists of sets of instructions that are executed at any given cycle whereas dynamically scheduled processor have a stream of single instructions. For example, a VLIW processor might be able to execute four operations concurrently. As not all applications are completely parallelizable, some cycles might have less operations at execution than the resources would permit. In addition, compilers that parallelize the input programs typically do not output the optimal parallelization but some suboptimal approximation.

2.3 Transport Triggered Architecture

When the the number of function units is increased to obtain more ILP, the interconnection network and the register file complexity of a VLIW processor grows exponentially [6]. This leads to excessive power consumption, substantial expansion of chip area, and decrease in achievable clock frequency. Transport Triggered Architecture is an exposed datapath architecture also capable of exploiting the static instruction-level parallelism [7]. TTA generalizes VLIW in the sense that the interconnection network is visible to the programmer, which even further moves complexity from the hardware to the compiler and alleviates the interconnection network bottleneck. In other words, the TTA processor template can be used to design VLIW type processors, but it is not merely limited for that purpose. Whereas VLIW is programmed using complete operations, TTA approach divides the operations into separate move operations. TTA processors are modular in a sense that the units can be compiled in any possible configuration, and the interconnection network allows an arbitrary

connectivity between the units.

The TTA interconnection network is formed of sockets that are attached to buses [7]. Sockets might be either unidirectional or bidirectional depending on the ports attached to them. Figure 2.1 shows an example TTA processor consisting of four different units: *Arithmetic Logical Unit* (ALU), *Load-Store Unit* (LSU), *Register File* (RF), and *Control Unit* (CU). Control unit is in charge of instruction fetching from instruction memory, instruction decoding, and the execution of control flow operations such as *call* and *jump*. Each port has a socket attached to it, and these sockets are further attached to the buses. Socket-bus connections are illustrated with the circles over the intersection of a bus and a socket. The arrows at the sockets indicate the data transfer direction.

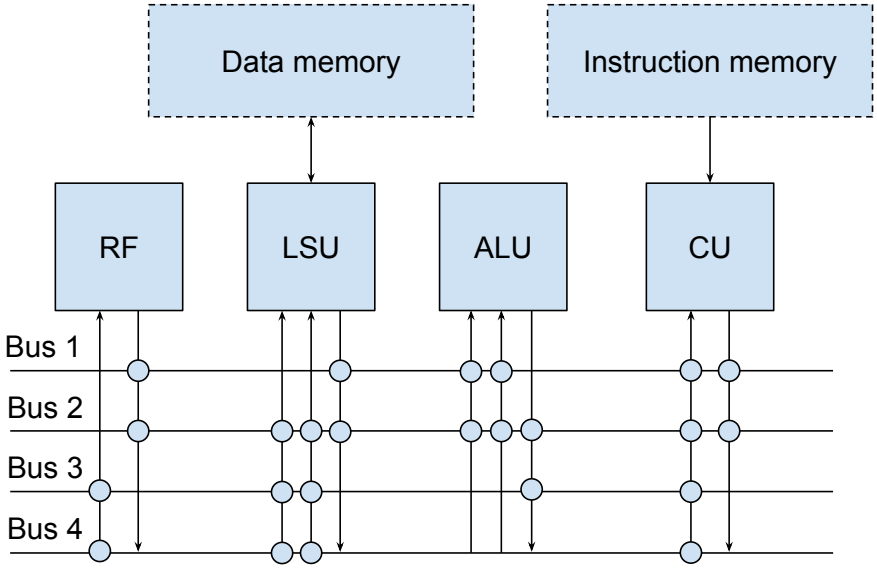

Figure 2.1: TTA processor with four buses and four units. Socket-bus connections are illustrated with the circles over the intersections. The arrows in the sockets indicate the data transfer direction.

2.3.1 Programming model

TTA programs consist of data transports, later referred to as *moves*, between the units on the interconnection network instead of the traditional way of programming complete operations [7]. A *bus* is capable of transferring a single value between two connected *sockets* on each cycle. A *program operation* is divided into *operand* and *result* moves. The operations are executed as side effect of the data transports into the units. More specifically, one of the operand moves is a trigger that starts the execution of an operation in a function unit. In addition, TTA processors are capable of transferring constant values, *immediates*, into the units on the buses. The

constant value is encoded into the move operation source field.

For example, an addition operation of two registers `ADD x,y` $\rightarrow$ `z` produces two operand moves `x` $\rightarrow$ `ALU.in` and `y` $\rightarrow$ `ALU.t`, and a single result move `ALU.out` $\rightarrow$ `z`. This naming convention is typical in TTA context: `in`(1...n) are input ports, `t` is the trigger port, and `out` (1...n) are output ports. Input operands of the addition operation are transported through the connections provided by the interconnection network to a compatible unit. After some operation latency, the result is read back to the register `z`. Operation latency depends on both the operation and the particular FU used.

An example of immediate value would be a subtraction operation of a register and a constant, `SUB 42,y` $\rightarrow$ `z`, which produces two operand moves `42` $\rightarrow$ `ALU.in` and `y` $\rightarrow$ `ALU.t`, and a single result move `ALU.out` $\rightarrow$ `z`. The value of 42 is transported directly to the `ALU` function unit.

2.3.2 TTA specific optimizations

The transport triggered architecture template allows optimizations not possible on other processor architectures [7]. *Register file bypassing* is an optimization that is traditionally performed at run-time by hardware. In case of a TTA processor, the compiler performs *software bypassing* at the compilation time.

In software bypassing, the compiler attempts to transfer results of an operation directly from FU to dependent operations passing the register. This alleviates register pressure and register file bottleneck by reducing the number of register accesses. Further, bypassing might eliminate some false dependencies between operations increasing the amount of scheduling freedom. Software bypassing may result in unnecessary register file write move after the result is transferred directly to all dependent operations. This removal of redundant result moves is called *Dead-result elimination* optimization. [7]

For example, consider a TTA program containing two dependent operations `ADD x,y` $\rightarrow$ `z` and `SUB z,a` $\rightarrow$ `b`. The latter operation uses the result of the former operation, thus the latter depends on the former. These subsequent moves produce the following TTA moves:

```
x → ADD.in
y → ADD.t
ADD.out → z
z → SUB.in
a → SUB.t
SUB.out → b
```

The result of the addition operation can be bypassed directly to `SUB.in`. After

the bypass, move `ADD.out -> z` can be eliminated as there are no other uses for the result. Figure 2.2 presents the possible data flow in the example. The dashed path represents the bypass case in which the result of the addition operation is directly written to `SUB.1`. The candidate move for dead-result elimination is highlighted by the dash-dotted line.

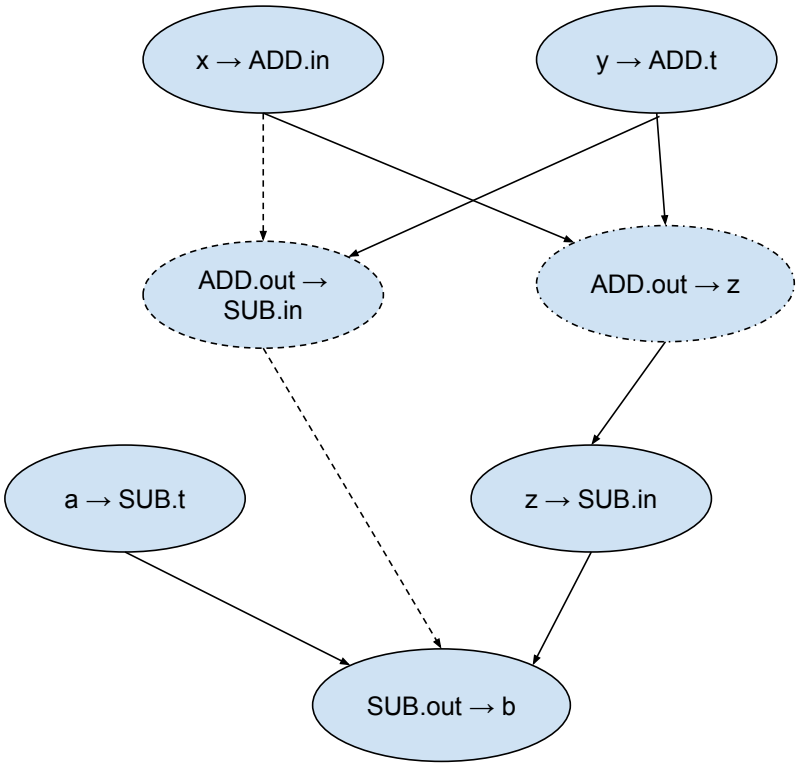

Figure 2.2: An example of a bypass opportunity. The bypass data flow is highlighted by the dashed line. The dash-dotted move node is a candidate for dead-result elimination.

2.3.3 TTA-based Co-design Environment

TTA-based Co-design Environment is a toolset developed at Tampere University of Technology for both designing and programming of synthesizable processors using the TTA template [8]. TCE attempts to provide a convenient set of tools for TTA processor design to minimize the time and the cost by automating as many steps as possible.

TCE provides multiple customization points for processor design including the interconnection network, the function units, and the register files. *Architecture Definition File* (ADF) describes a TTA processor implementation. The file lists all units that the processor contains, and how the interconnection network combines the units together. In other words, a TTA processor in TCE is constructed from an arbitrary set of units, and any kind of interconnection network that attach the units together.

In TCE, processor design typically starts off with a high level language program and a set of performance requirements and design limitations that must be met. Once the high level language program has been implemented and tested natively, a suitable TTA processor is constructed of function units, register files, and an arbitrary connectivity network. The program is compiled to the designed processor and simulated using a simulator, and the simulator metrics are used to evaluate the design. This iterative process is carried out until all requirements have been met. Below are listed the main tools of the TCE toolset that are involved in a processor design.

Processor Designer (ProDe) is a graphical processor architecture design program [8]. The tool provides an interface for graphically altering the designed TTA processor, for example adding new units, and editing the interconnection network. The designed processor is serialized into an ADF-file. *Processor Generator* (ProGe) takes the output from the processor designer and generates a synthesizable *Hardware Description Language* (HDL) description of the processor.

The retargetable TCE compiler *tcecc* compiles programs written in `C`, OpenCL `C`, and `C++` programming languages [8]. The compiler targets either TCE-specific sequential bitcode or an architecture specific parallel TTA program in the *TTA Program Exchange Format* (TPEF) as the low level language.

In order to verify and benchmark a designed processor architecture, the program execution is typically simulated using a software that models the processor [8]. TCE contains *TTA Simulator* (ttasim) that cycle-accurately interprets the input program on an architectural model of the processor. The simulator requires the architecture definition file of the simulated processor, and a program that is compiled to TPEF against the ADF-file.

Program Image Generator (PIG) converts the compiled TPEF program into instruction memory and data memory images [8]. These images may be used along with the generated HDL environment to test the processor in HDL simulators, and for programming the fabricated final processor.

3. COMPILERS

A compiler is a computer program that transforms source code written in a source language into another computer language, the target language. Usually the compiler translates a high-level programming language to a lower level language such as assembly language. The compiler *analyses* the high-level language statements of the source program and transforms them into an internal representation that is further processed. The internal representation, given that it is not syntactically ill formed or semantically invalid, is *synthesized* into a target program.

Modern compilers are typically organized around multiple *passes* which are successive stages in the compilation [9]. The phases of the compilation transform the representation of the source program to another. The advantage of *multi-pass compilers* is the possibility to perform many sophisticated optimizations. Also the input to one optimization may depend on the output of another optimization. In addition, a compiler that consists of multiple small parts is more understandable, and simpler to test and develop.

A compiler is typically divided into three separate parts: the front-end, the middle-end, and the back-end [9]. The front-end is in charge of analyzing the high-level language programs and generating *an intermediate program representation* (IR). Typically compilers have multiple front-ends for different high-level programming languages.

The middle-end performs language- and machine-independent optimizations on the intermediate program representation. These optimizations may include for example removal of useless or unreachable operations, and function inlining. Different front-ends and back-ends share a common middle-end which allows different high-level language compilers targeting various architectures have shared implementation.

The back-end is aware of the target machine architecture and in charge of generating the output language code. Typically the output language is the native machine language of the target machine. In addition, there are numerous machine-dependent optimizations that are performed in this phase. Figure 3.1 presents the structure of a typical compiler.

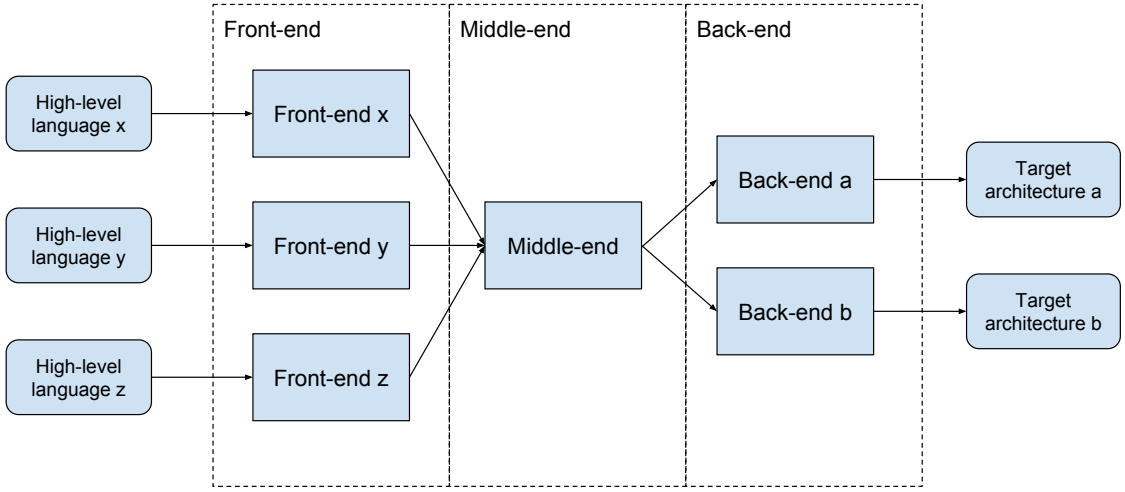

Figure 3.1: The structure of a typical compiler.

3.1 Intermediate Representation

The front-end of a compiler transforms the high-level language to an intermediate representation of the input program [9]. The intermediate representation is implemented with intermediate language that is targeting an abstract virtual machine. The abstract machine has a virtual operation set of primitive operations that typically are found in most processor architectures. The IR stores the data in virtual registers of the abstract machine. The number of available virtual registers is usually higher than that of a target machine, or even infinite.

Using intermediate representation allows modular and clear separation of different compiler parts. This way a single compiler might target multiple input languages, and similarly support code generation for multiple target architectures.

3.1.1 Static Single Assignment Form

Static Single Assignment (SSA) is an intermediate representation where each virtual variable is assigned only once [10]. If a program variable is assigned multiple times, a new version of the variable is created for each assignment. This IR design simplifies the analysis of variables.

For example, Figure 3.2 shows a high-level language program and its corresponding SSA form. The transformation is done by splitting variable pos into variables pos_0 and pos_1 as shown in Figure 3.2 (b). Function φ is used to merge the different values of a variable at a join point. In this case, the merged value of pos_0 and pos_1 is assigned to pos_2.

```
x = f();
if x > 0 then
    pos = true;
else
    pos = false;
print(x, pos);
```
(a) High-level language

```
x0 = f();
if x0 > 0 then
    pos0 = true;
else
    pos1 = false;
pos2 = φ(pos0, pos1)
print(x0, pos2);
```
(b) SSA form

Figure 3.2: High-level language program before and after SSA transformation.

3.1.2 Data Dependence Graph

Data Dependence Graph (DDG) is a directed acyclic graph $G = (V, E)$, where V is a set of nodes, E is a set of edges, which are ordered two-element subsets of V. The nodes correspond to operations of the program and edges to the dependencies between the operations. The weights of the edges resemble the minimum delay interval between the execution of the predecessor and successor operations of the edge. The graph is utilized whenever the ordering of operations is considered. For example, instruction scheduling relies on the information that the DDG presents.

Critical path of a DDG is the longest directed path between any root node and leaf node. However, the critical path is not unique because multiple paths in the graph may have an equal length. The length of the critical path lays a lower bound of cycles to the schedule. It might be impossible to achieve the optimal schedule due to limitations set by the utilized hardware.

Data dependence analysis considers the ordering constraints between the operations, and results in a data dependence graph of the program. The constraints limit the amount of ILP that one can exploit. There are two kinds of data dependencies, *flow dependencies* and *anti dependencies*. Flow dependencies occur due to data being shared by variables whereas anti dependencies rise when operations share a read register. Anti dependencies are sometimes called false dependencies because they are caused by register re-usage and not by the program semantics. Data dependence analysis determines the limits in which the operations can be reordered and parallelized.

Read after Write (RaW) is a dependency where a result written to register is later used. For example, consider two operations, `ADD x,y` $\rightarrow$ `z` and `SUB z,a` $\rightarrow$ `b`. As the substitution operation uses register `z`, there is a RaW dependency between the operations.

When a register is being written multiple times, write operations are *Write after*

Write (WaW) dependent. For example, two addition operations `ADD x,y → z` and `ADD a,b → z` that write the result to same register are WAW dependent.

Write after Read (WaR) is anti dependency between two operations. If a register is used after a read, the write and the read operations must be WAR dependent to produce correct result. For example, two addition operations `ADD x,y → z` and `ADD a,b → y` are WaR dependent because the latter operation rewrites the register used by the former operation.

In addition to the dependencies listed above, operations also cause dependencies between the generated moves in TTA processors. The result moves of an operation depend on the input operands. For example, consider operation `ADD x,y → z` and the generated moves:

```
x → ADD.1
y → ADD.2
ADD.3 → z
```

Move `ADD.3 → z` must depend on the input operands `x → ADD.1` and `y → ADD.2` to preserve correct execution order of the operation stages.

The outcome of data dependence analysis is the data dependence graph. For example, two operations, `ADD x,y → z` and `SUB z,a → z` produce the following TTA moves:

```
x → ADD.1
y → ADD.2
ADD.3 → z
z → SUB.1
a →  SUB.2
SUB.3 → z
```

These moves form a DDG as shown in Figure 3.3, assuming that addition and subtraction operations take two cycles, and a register read consumes one cycle. The edges are labeled with the dependence types and the edge weights, and the critical path is highlighted by the dashed line.

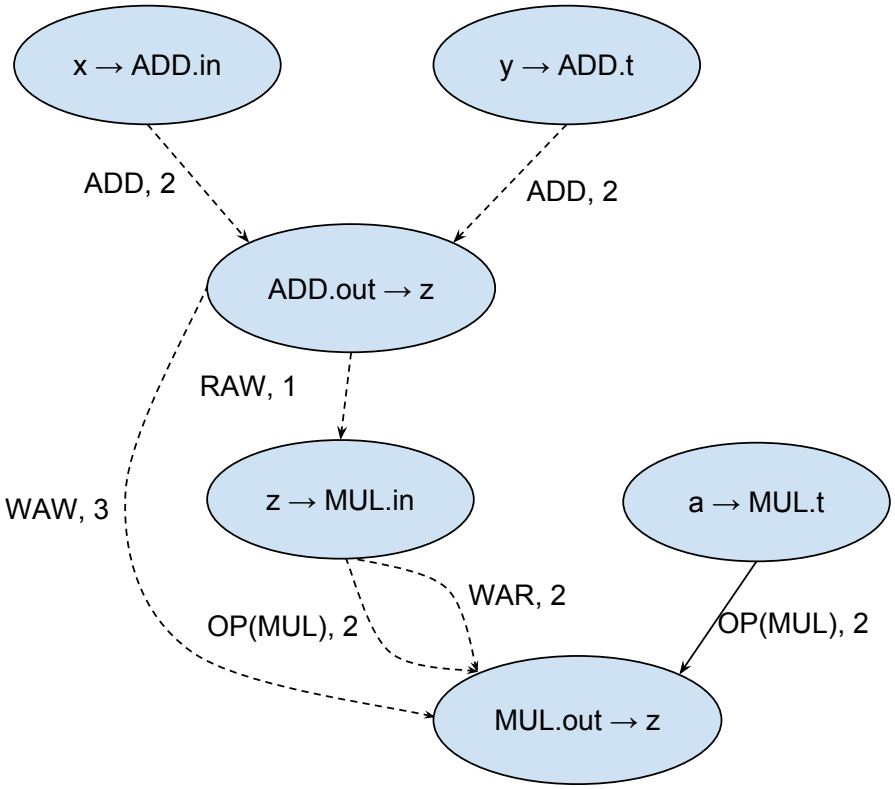

Figure 3.3: Data dependence graph of two operations, `ADD x,y` → `z` and `MUL z,a` → `z`. Edges are labeled with the dependence types and the edge weights. The critical path is highlighted by the dashed line.

3.2 Code Generation

Code generation is the transformation of the intermediate representation into the instruction set of the target processor performed by the backend. The major tasks in code generation are instruction selection, register allocation, and instruction scheduling. The flow of compilation is illustrated in Figure 3.4.

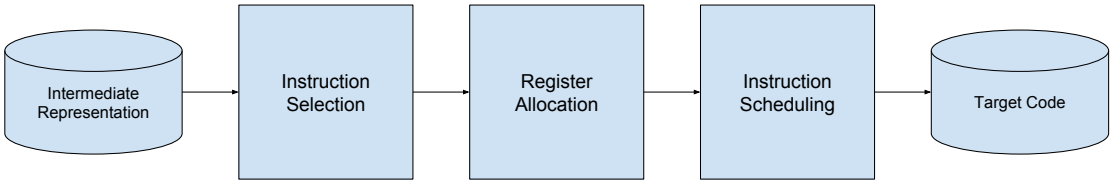

Figure 3.4: Code generation process from the intermediate representation to the target code.

Code generation for the transport triggered architecture is in many ways identical to that of common general purpose processors. The main differences are the

requirement for statical compile-time scheduling, which this thesis concerns, and various TTA-specific optimizations.

The subsections below describe the fundamental components involved in the compilation of a program. The example program below is used to illustrate the code generation process step by step.

```
z = ++x + y
b = z - a
```

In the example, at first the variable `x` is incremented by one using the increment operation of the high-level language. The result of increment operation is then added to `y`. At last, this intermediate result is subtracted by `a` and stored to variable `b`.

3.2.1 Instruction Selection

Instruction selection is a process of mapping virtual IR operations to target processor instructions [11]. It is quite common that a single target instruction covers a set of IR instructions, and that a IR operation is expanded to a sequence of target machine operations. In practice, a compiler attempts to find a set of non-overlapping instruction templates that cover all the IR instructions. Instruction selection is proven to be NP-complete.

An instruction selector typically operates on a graph of the input program. For example, *SSA graph*, which is a graphical representation of the input program in SSA form, could be used in instruction selection. There are usually multiple ways in which the graph can be covered with instruction templates. The compiler might for example attempt to minimize the number of target instructions, or optimize for some other cost over the target instructions.

Below are the TTA moves that the example above results in after the instruction selection. The increment operation is executed using an addition by a constant of one. The addition and subtraction are carried out with corresponding operations.

```
x → ADD.in
1 → ADD.t
ADD.out → x
x → ADD.in
y → ADD.t
ADD.out → z
z → SUB.in
a → SUB.t
SUB.out → b
```

Consider the increment operation for variable `x`, `++x`. This operation can be covered with the `ADD` instruction as shown above. The variable `x` is transported into a function unit that contains the operation `ADD`, incremented by the constant of one, and stored back to the variable. Another way to cover the increment operation with machine instructions is to use the increment operation that is found in many modern architectures instead of the `ADD` instruction. The `INC` operation increments a register by one and is typically notably faster the addition operation. More specifically, in case of a TTA processor there then would be no need to transport the constant value one to the function unit.

3.2.2 Register Allocation

The intermediate representation operates on virtual registers or variables. *Register Allocation* is a process of mapping these virtual variables to real registers in the target processor [12]. Each virtual variable has a *live range*, a time interval between the first and the last usage of the variable. The live range of a variable determines when a given variable needs to exists in either a register or memory. After the live range, a variable is announced dead, meaning that it is not used anymore.

The number of concurrent live ranges of variables might exceed the number of physical registers in the hardware. This raises the need for *spilling* values to memory. A special *spill code* needs to be inserted in appropriate locations of the program to store live variables in memory. Similarly, the variables need to be fetched back to registers when they are needed again. Random accessible memory can be an order of magnitude slower than processor registers. For this reason the task of register allocator is to minimize register spilling and to keep the frequently used variables in registers. Register allocation problem is proven to be NP-complete [13].

The re-usage of registers introduces anti dependencies between operations that reduce the change for concurrency [14]. These dependencies are name dependencies, that is, they can be removed by renaming the registers used. This optimization is known as the *register renaming*.

Continuing with the example, below is a listing of the source code with the registers assigned. The notation `RF.n` means that the variable is assigned to nth register of register file `RF`. Register `RF.0` is reused when the result of the addition operation is transported back to the register file `RF`.

```
RF.0 → ADD.in
1 → ADD.t
ADD.out → RF.0
RF.0 → ADD.in
RF.1 → ADD.t
```

```
ADD.out → RF.0
RF.0 → SUB.in
RF.3 → SUB.t
SUB.out → RF.4
```

3.2.3 Instruction scheduling

Instruction scheduling is a compiler optimization that attempts to reorganize the execution order of instructions to improve performance. In dynamically scheduled processors, instruction scheduling rearranges operations to improve the utilization of processor resources. Operations consists of multiple stages in the processor, and typically these stages can be interleaved. Pipelining might introduce *hazards* due to dependencies between operations. Hazards result in *pipeline stalls*, *No Operation* (NOP) executions until the dependency is satisfied. During the hazards, the instruction scheduler can appoint other instructions from the pipeline that are not dependent of the current result.

Instruction scheduling is particularly important in TTA processors for efficient execution. TTA processors typically have multiple concurrent function units that are able to take advantage of pipelining. A TTA instruction scheduler takes account of all available resources on the processor and attempts to utilize as much ILP as possible. The instruction scheduler assigns operations to function units, and moves to transport buses and sockets.

One possible schedule for the TTA moves of the example above, given the architecture shown in Figure 2.1, is presented in Table 3.1. It is assumed that the addition operation takes two cycles and the subtraction operation three cycles. All operations are assigned to the function unit ALU.

Cycle	Bus 1	Bus 2	Bus 3	Bus 4
1	RF.0 → ALU.in	1 → ALU.t	-	-
2	-	-	-	-
3	-	-	ALU.out → RF.0	-
4	RF.0 → ALU.in	RF.1 → ALU.t	-	-
5	-	-	-	-
6	-	-	ALU.out → RF.0	-
7	RF.0 → ALU.in	RF.3 → ALU.t	-	-
8	-	-	-	-
9	-	-	-	-
10	-	-	ALU.out → RF.4	-

Table 3.1: A schedule for the TTA moves presented in the example above for the TTA processor shown in Figure 2.1.

Figure 3.5 shows the moves assigned at cycle 4. The dashed-line path represents

the move `RF.0` → `ADD.in` and the dashed-dotted line move `RF.1` → `ADD.t`. The socket to bus connections highlighted with red color are the ones connected at this cycle.

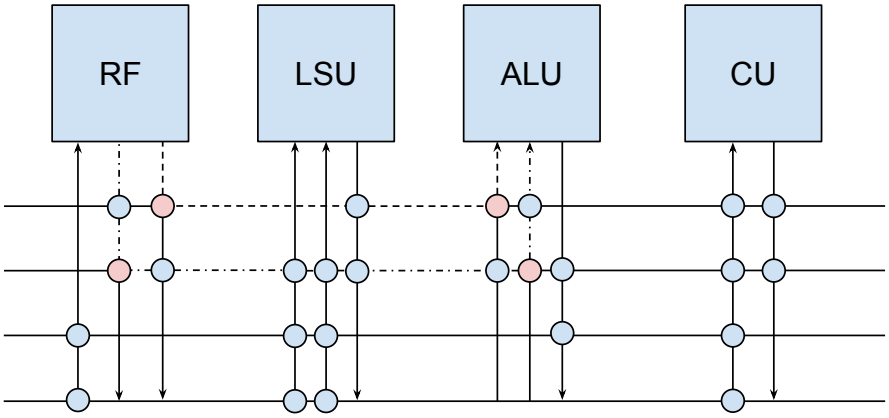

Figure 3.5: An illustration of the schedule shown in Table 3.1 at cycle 4.

The result is read from `ALU` at cycle 6 as is shown in Figure 3.6. The `SUB` operation move assignments are identical to those of the addition operation.

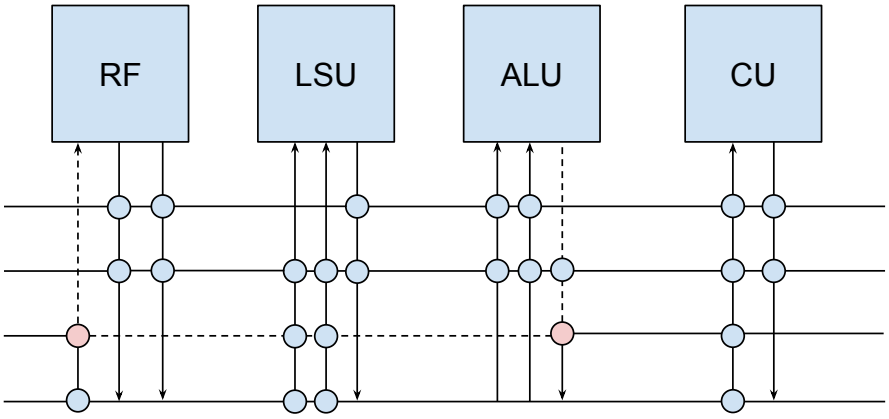

Figure 3.6: An illustration of the schedule shown in Table 3.1 at cycle 6.

3.2.4 Phase Ordering

Register allocation may be performed either before, after, or at the same time as instruction scheduling. Register allocation attempts to minimize the spilled variables and reuse registers as much as possible. Reuse of registers leads to a higher number of dependencies between operations that the instruction scheduler must cope with. Register renaming during scheduling reduces the number of these additional dependencies. On the other hand, the instruction scheduler attempts to

exploit as much ILP as possible, given the particular TTA processor, leading to more spilling as the number of physical registers is typically much lower than the amount of virtual registers alive in parallel. These two objectives are in conflict with each other. Simultaneously performed register allocation and instruction scheduling require complicated algorithms and makes the scheduler monolithic.

3.3 TCE Retargetable Compiler Structure

The TCE retargetable compiler is a high-level language compiler targeting *C*, *C++*, and *OpenCL C* programming languages [8]. The front-end is implemented around the *Clang* compiler front-end that is part of the *LLVM compiler infrastructure* [15]. The Clang compiler produces intermediate representation named *LLVM intermediate representation*. The LLVM intermediate representation is in SSA form. TCE utilizes the numerous middle-end optimizations that LLVM provides around the LLVM IR language. [15] The back-end of the TCE compiler is customized specifically for TTA processors. Figure 3.7 presents the structure of the TCE compiler.

In tcecc, the instruction scheduler operates on a single *basic block*, which is a set of operations that have only one entry point and only one exit point. These types of schedulers are known as *local schedulers*. In addition, register allocation is done before instruction scheduling to reinforce modularity. This design choice prioritizes efficient register usage over exploiting all possible ILP. Schedule-time register renaming is used to compensate the disadvantages of this approach in tcecc.

This thesis proposes a new instruction scheduling algorithm for the TCE compiler, highlighted in the Figure 3.7. The scheduler is implemented as a pass to the TTA specific back-end of the compiler. This pass gets a basic block and the current TTA processor configuration, that is the units and the interconnection network, as an input, and results in a set of scheduled moves as an output.

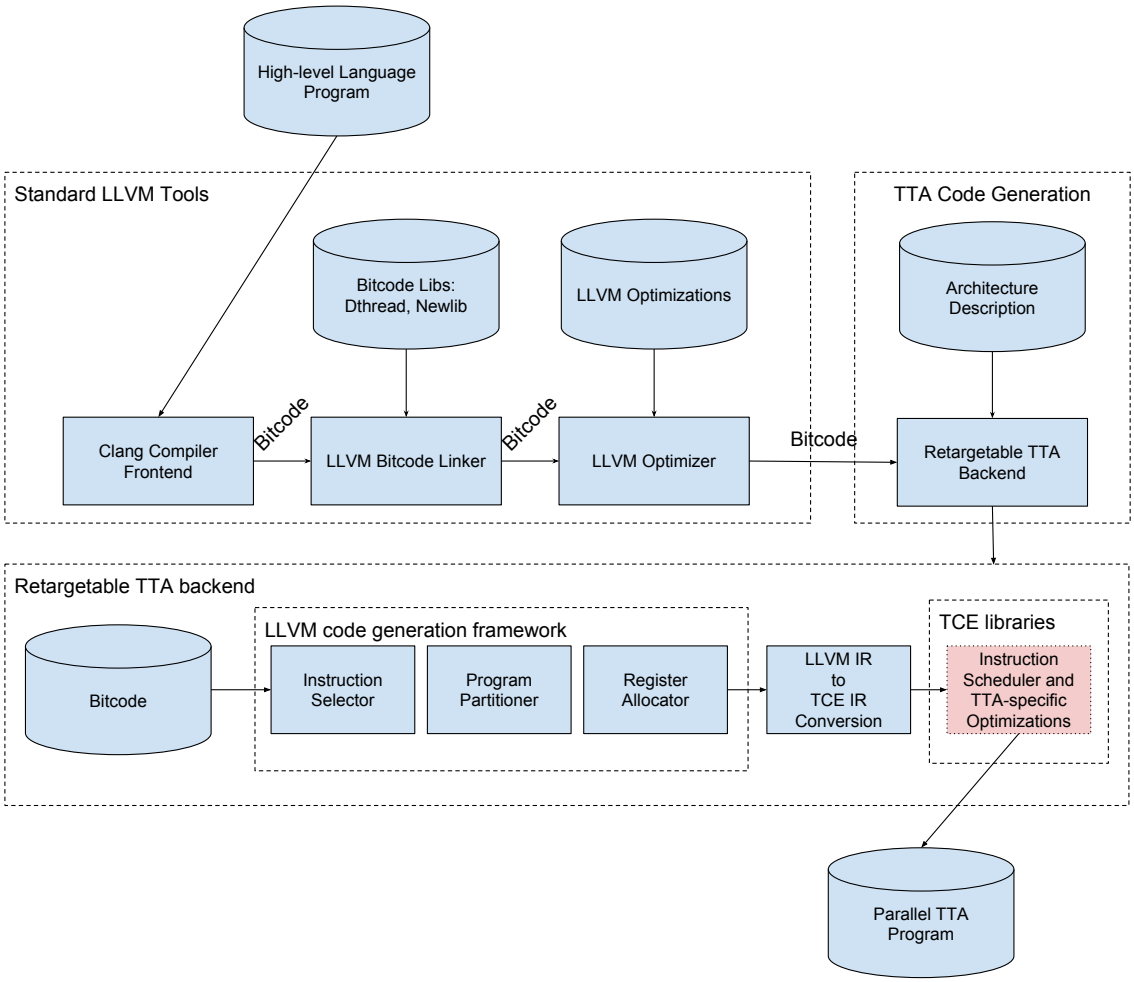

Figure 3.7: Overview of the TCE compiler. The highlighted part indicates the modified component in this thesis project.

4. INTEGER LINEAR PROGRAMMING

A *Constraint Satisfaction Problem* (CSP) $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, D, C \rangle$, where X is an n-tuple of variables $X = \langle x_1, x_2, ..., x_n \rangle$, D is an n-tuple of domains $D = \langle D_1, D_2, ..., D_n \rangle$ and C is an m-tuple of constraints $C = \langle C_1, C_2, ..., C_m \rangle$. Each variable x_i is in the corresponding domain, $x_i \in D_i$. A constraint C_j is a pair $\langle X, S_j \rangle$, where S_j is a relation on the variables [16].

A *feasible solution* is an assignment x of values to the variables in X, where the assigned values appear in their respective domains D, that satisfies the constraints in C. There might be no solution, a single solution, or multiple solutions. Typically the applications are *under constrained*, that is they have multiple solutions which can be arranged according to some property of the solution. These kinds of problems are called *Constraint Optimization Problems* (COP). In addition to the constraints, they have an objective function f which is to be either minimized or maximized. An *optimal solution* x^* is a feasible solution such that for any other feasible solution x', $f(x^*) \leq f(x')$ given that the objective function is to be minimized.

Integer Linear Programming problems are a subset of constraint satiscation problems in which the variables X are restricted to be integers, and the objective function f and the constraints D are linear. Moreover, 0-1 integer programming is a special case of integer linear programming problems where variables are required to be binary, that is, 0 or 1. The practical applications of integer linear programming problems are numerous including scheduling, artificial intelligence, and resource allocation [17].

P is the set of all problems that can be solved in polynomial time by a deterministic Turing machine. Nondeterministic polynomial time (NP) problems are a set of all computational problems that have efficiently verifiable solutions, that is they are solvable in polynomial time by a nondeterministic Turing machine (NDTM). Since all deterministic Turing machines are a subset of nondeterministic Turing machines, P is also a subset of NP. Stephen Cook introduced the theorem of NP-completeness through his famous 1971 paper entitled 'The Complexity of Theorem Proving Procedures' [18]. The prevailing belief is that NP-complete problems do not have polynomial time algorithms, and thus are not in P [19]. A decision problem S is NP-complete if the feasibility of any given potential solution can be verified in polynomial time and if any NP problem can be Turing-reduced to S in polyno-

mial time [18]. Thus, if there exists a polynomial time algorithm for a NP-complete problem S, all other NP-complete problems can be solved in polynomial time by first converting them to an instance of the problem S. Consequently, NP-complete problems are the most important problems amongst NP problems.

Constraint satisfaction problems are typically in NP [20], and many important constraint satisfaction problems are proven to be NP-complete [16]. In particular, integer linear programming has been shown to be NP-hard [21] and 0-1 integer programming to be NP-complete [22].

4.1 Modeling Optimization Problems with Integer Linear Programming

There are useful common patterns that can be utilized to model the relations of problem variables. The knowledge of these patterns also helps understanding complex integer linear programming constraints. Below we list some common ways to model conditional variable relations with linear constraints. In these examples, all variables are binary.

In scheduling, some one-time events might be required to happen at least n units of time before others. This can be modeled as follows: given a prerequisite event x_t and a set of dependees $Y = \{y_t\}$ ($t \in [0, T]$), the following must hold:

$$x_t + \sum_{y \in Y} \sum_{t'=0}^{t+n} y_{t'} \leq 1, \text{ for } t \in [0, T]. \tag{4.1}$$

This constraint requires that if x_t equals 1, then none of the variables in the set of dependees can be equal to 1 within the time interval $[0, t + n]$. Consequently, x_t must happen at least n units of time before the dependee events. For example, when scheduling a manufacturing process consisting multiple steps, consecutive steps should be assigned to correct order and there must be the delay of each subprocess in between the steps.

Similarly, a one-time event might enforce a set of other events to happen within n units of time of its occurrence. To constraint the set of other events to occur at most n units of time before x_t, the following most hold

$$x_t - \frac{1}{|Y|} \sum_{t'=t-n}^{t} \sum_{y_t \in Y} y_t \leq 0, \text{ for } t \in [0, T]. \tag{4.2}$$

Whereas if the event constrains the set of other events to happen at most n units of time after x_t,

$$x_t - \frac{1}{|Y|} \sum_{t'=t}^{t+n} \sum_{y_t \in Y} y_t \leq 0, \text{ for } t \in [0, T]. \tag{4.3}$$

These constraints require that all the binary variables in the set Y are equal to one exactly once within the corresponding time interval if and only if the variable x_t equals 1. This causes the summation to equal the cardinality of the set Y and thus the subtraction equals zero. In case the prequisite variable equals zero, the values of variables in the set Y are not constrained. For example, a manufacturing process might require that some consecutive steps must be done within predetermined time interval.

Mutually exclusive variables can be precluded by constraining the sum of the variables be equal to one. Similarly, the mutual exclusivity of sets of variables leads to

$$\sum_{X \in S} \frac{1}{|X|} \sum_{x \in X} x = 1, \tag{4.4}$$

where S is a set of all mutually exclusive sets of variables and $|X|$ is the cardinality of the set X. This equation constraints that exactly one set has a variable of value 1. By varying the scale factor $|X|^{-1}$, it is possible to set different bounds to the variables in sets. A scale factor of 0.5 allows a single set to have exactly two variables that have a value of 1. For example, there might be different alternative processes consisting of independent steps in a manufacturing process of which a single process is to be chosen.

4.2 Example models

4.2.1 Knapsack

The knapsack problem is a problem of constraint optimization. The objective is to determine a collection of items so as to maximize the total value of the knapsack without exceeding the capacity of the knapsack. The knapsack problem is NP-complete [23]. The knapsack problems represent a very large number of real-world problems and often arises in economics, financial optimization, optimization of available resources, and cryptography.

Knapsack problem can be modeled as an integer linear programming problem. Let an integer variable x_i be the quantity of each item, $X = \langle x_1, x_2, ..., x_n \rangle$. For each item, the domain is $D_i = \{d \in \mathbb{N} \mid 0 \leq d \leq d_i\}$, where d_i is the largest number of items that weight less than the capacity of the sack such that $d_i w_i \leq C$, in which w_i is the weight of the ith item and C is the capacity of the sack.

The overall weight cannot exceed the knapsack capacity, yielding constraint $\sum_{i=1}^{n} w_i x_i \leq C$. In order to find an optimal collection of items, objective function of $\sum_{i=1}^{n} x_i v_i$, where $v_i \geq 0$ is the value of the ith item, is to be maximized.

4.2.2 Traveling salesman

The *Traveling Salesman Problem* (TSP) is the one of the most intensively studied combinatorial optimization problems. It has been proven to be NP-complete [24]. It is simple to describe and yet computationally incredibly hard: for m cities and their pairwise cost of traveling $c_{ij} \geq 0$, determine the most economic way of visiting each city exactly once. The practical applications of the TSP include scheduling, logistics, and microchip manufacturing [25, 26].

The TSP can be formulated as a 0 – 1 integer programming problem. Let x_{ij} be a boolean variable representing whether the journey from ith city to jth is included in the route, indicated by a true value. This produces $m-1$ variables for each city and $m(m-1)$ variables altogether. The salesman must pass through each city once:

$$\sum_{j=1}^{m} x_{ij} = \sum_{i=1}^{m} x_{ij} = 1, \text{ for } j = 1, ..., m, i = 1, ..., m.$$

This constraint might lead to infeasible solutions because it is possible to route the salesman through disjoint subtours instead of a single trip. Consequently, additional constraint, $\sum_{i \in K} \sum_{j \in K} x_{ij} \leq |K| - 1$, for all $K \subset \{1, ..., m\}$, must be included in order to eliminate the subtour solutions. This inequality ensures that the route must leave each subset K. The objective function to be minimized is $\sum_{j=1}^{m} \sum_{i=1}^{m} c_{ij} x_{ij}$.

4.3 Solving Integer Linear Problems

Constraint satisfaction problems are typically solved using some sort of search exploring all the possible assignments of the variables. Typical techniques are backtracking and constraint propagation [16]. There are numerous algorithms to solve integer linear problems exactly. Following subsection presents an algorithm called *Branch and Bound* (BB).

4.3.1 Branch and Bound Algorithm

Branch and bound algorithm is a two-stage algorithm that is used for a number of NP-hard problems [27]. *Branching* step splits the problem into two or more subproblems, whereas *bounding* results in calculated lower or upper bounds for the objective function value of the subproblem. These bounds are used to prune unpromising subproblems. The problem is solved when there exist a single solution or when the upper bound matches the lower bound. This leads to a search tree whose nodes are the subproblems of the problem.

Linear programming relaxation (LP) is a solving technique where all integer constrains are replaced with their continuous counterparts [28]. For example, a binary variable $x \in \{0, 1\}$ would be replaced with a continuous variable $x \in [0, 1]$. The

resulting relaxation is a linear program that is solvable in polynomial time. That is to say, LP-relaxation transforms a NP-hard optimization problem into an analogous problem that can be solved in polynomial time to gain information about the solution to the original problem. LP-relaxation is typically used in branch and bound to find the bounds of each subproblem.

For 0-1 integer linear programming, branch and bound with LP-relaxation is performed as follows. The branching is done by dividing the problem into two subproblems, one in which the variable is set to 0 and the other in which the variable is set to 1. The solution to the LP-relaxation provides a bound for this subtree; If the current solution is worse than the best integer solution found so far, the solver backtracks from this subtree. Otherwise, the subtree is considered further. If the current node is a leaf node of the search tree, the integer solution is checked against the best integer solution so far. This procedure is recursively applied until the search tree has been examined entirely.

For example, consider the following binary integer linear programming problem:

$$\begin{aligned} &\text{Maximize:} \\ &9a + 5b + 6c + 4d \\ &\text{Subject to:} \\ &6a + 3b + 5c + 2d \leq 10 \\ &c + d \leq 1 \\ &-a + c \leq 0 \\ &-b + d \leq 0 \end{aligned} \tag{4.5}$$

The optimal solution to the LP-relaxation $a, b, c, d \in [0, 1]$ is $(5/6, 1, 0, 1)$, and the corresponding objective value equals 16.5. The first branch is done over variable a. Let consider the branch in which a equals 0, where the problem becomes the following

$$\begin{aligned} &\text{Maximize:} \\ &5b + 4d \\ &\text{Subject to:} \\ &3b + 2d \leq 10 \\ &-b + d \leq 0. \end{aligned}$$

Since a equal to zero, c must be zero as well due to constraint $-a + c \leq 0$. The constraint $c + d \leq 1$ can be eliminated also as d is in any case less than 1. The optimal solution to the LP-relaxation of this subproblem is $(0, 1, 0, 1)$ with an

objective value of 9. This becomes the current best known solution because all the variables are binary valued.

The LP-relaxation of the branch where a equals 1 becomes

$$\begin{aligned} &\text{Maximize:} \\ &9 + 5b + 6c + 4d \\ &\text{Subject to:} \\ &3b + 5c + 2d \leq 4 \\ &c + d \leq 1 \\ &-b + d \leq 0. \end{aligned}$$

The subproblem has an optimal solution at $(1, 0.8, 0, 0.8)$ with an objective value of 16.2. As the objective value is greater than the current best known binary solution, the branch is worth exploring further. Next, branching is done over variable c. Setting b to equal 0 produces the following subproblem

$$\begin{aligned} &\text{Maximize:} \\ &9 + 6c \\ &\text{Subject to:} \\ &5c \leq 4 \end{aligned}$$

The optimal solution to the LP-relaxation of the subproblem is $(1, 0, 0, 0)$ with an objective value of 9. The algorithm backtracks from this subtree as the objective value is less than that of the current known best solution.

When c equals 1, the constraint $3b + 5c + 2d \leq 4$ becomes $3b + 2d \leq -1$ and is thus unfeasible.

Next, the other branch over b, in which the variable equals 1, is considered. The optimal solution to the LP-relaxation of the subproblem

$$\begin{aligned} &\text{Maximize:} \\ &14 + 6c + 4d \\ &\text{Subject to:} \\ &5c + 2d \leq 1 \\ &d \leq 1 \end{aligned}$$

equals $(1, 1, 0, 0.5)$. The objective value equals 16, which is higher than the current known best binary solution. As c is already 0 in the parent LP-relaxation solution, the same solution is the best solution of this branch as well. Setting d to 0 gives an objective value of 14 and becomes the current best known binary solution. The solution where d equals 1, that is $(1, 1, 0, 1)$, is not feasible as the constraint $5c+2d \leq 1$ is not satisfied.

When c equals one, the solution is not feasible. Therefore $(1, 1, 0, 0)$ is the optimal solution to the problem with an objective value of 14. The search tree of this branch and bound search is illustrated in Figure 4.1.

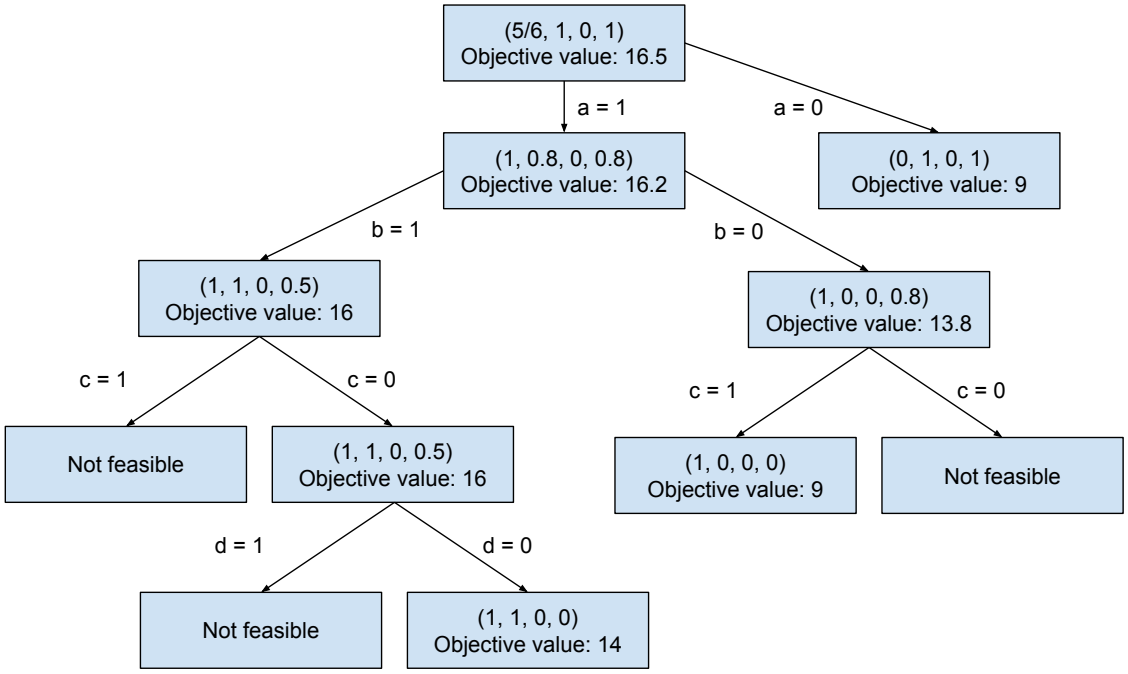

Figure 4.1: Search tree of the integer linear problem described in Equation 4.5.

4.3.2 Special Ordered Sets

A *Special Ordered Set* (SOS) is an ordered subset of model variables to specify integrality conditions [29]. Branch and bound algorithm may work more intelligently knowing that a variable belongs to an ordered set of variables. The branching order of variables is prioritized as the special ordered set dictates. For example, in scheduling there are typically sets of mutually exclusive variables and some of the variables are more preferable than others.

There are two kinds of special ordered sets. *Special Ordered Sets of type One* (SOS1) are defined to be an ordered set of variables at most one of which may be non-zero in a feasible solution. Typically an SOS1 is to represent a set of mutually exclusive alternatives ordered in increasing cost to guide the search order. *Special Ordered Sets of type Two* (SOS2) is a set of consecutive variables in which not more than two adjacent members can be positive, all others being 0.

5. INTEGER LINEAR PROGRAMMING FORMULATION OF THE TTA INSTRUCTION SCHEDULING PROBLEM

Let us model the TTA instruction scheduling problem as $0-1$ integer linear programming problem. The scheduler input consists of the sequential operation moves generated from the compiled program, and the data dependency graph of the moves. As register allocation is currently done prior to scheduling, move's source or destination might be already determined. After scheduling, all moves are assigned to the interconnection network onto a single cycle, and consequently the operations are appointed to function units.

For each move there may exist multiple destination and source ports, and similarly there might be more than one connection between a pair of destination and source port on the interconnection network. Immediate values only have a destination port and an assigned bus. In addition to the moves generated from the compiled program, an additional bypass move is created for each result move that is candidate for bypass optimization given the interconnection network. This move, the *bypass move*, is generated by taking the source from the *bypass candidate* move and the destination from the *bypass result* move.

The decision variables of the model specify the connection to be used, and the cycle the move is to be executed. Associate binary variable

$$M_{i,t,c} = \begin{cases} 1 & \text{Move } i \text{ is assigned to connection } c \text{ at cycle } t, \\ 0 & \text{otherwise,} \end{cases}$$

to describe a possible move assignment, indicated by a true value.

The interconnection network confines a set of possible connections C_i for each move i. For each move there exist variables for all possible connections C_i, and cycles in the range $[t_{min}, t_{max}]$. Let P be the set of all moves of a considered TTA program.

5.1 Completeness

This section summarizes the aspects of the modeled problem that must be taken into account. For each requirement, there is a reference to the subsection that proposes the constraints that take care of the particular prequisite. The purpose is to provide an informal completeness analysis of the integer linear programming model.

- Each move must be assigned exactly once, as proposed in Subsection 5.2.1. Exception to this are bypass related moves which have more complicated relations and mutually exclusive options. Bypass related constraints are outlined in Subsection 5.2.3. Above-mentioned constraints produce a schedule in which all the moves are assigned but different resource limitations are not taken into account.

- Some operations require results from previous operations, but preceding constraints allow operations to be assigned at an arbitrary order. In order to obey these dependencies, Subsection 5.2.2 proposes constraints that restrain the ordering to adhere with the dependencies.

- Multiple moves might be assigned to same bus at a given cycle. Subsection 5.3.3 lays out constraints to enforce that each bus transport only a single move at each cycle.

- Function units can start an execution of a single operation at a cycle. Moreover, pipelined operations should be interleaved so that none of the results are being overwritten before they are transported to the corresponding destinations. Subsection 5.2.5 presents the necessary constraints.

- Register files might be accessed more than once per cycle per port. The access is bounded by the number of input and output ports attached to the unit. Each port can transport a single value at a cycle. Register file constraints are proposed in Subsection 5.2.4.

5.2 Constraints

5.2.1 All Moves Must be Assigned

All moves $M_{i,c,t}$ that are not bypass moves, moves candidate for bypassing, or bypass result moves must be assigned exactly once to a connection and a cycle:

$$\forall i : \sum_{c \in C_i} \sum_{t=t_{min}}^{t_{max}} M_{i,c,t} = 1. \tag{5.1}$$

5.2.2 Dependencies Between Moves

The data dependency graph sets an order of execution for moves of a given program. Let $M_{i,c,t}$ be an indicator associated with an arbitrary move and $M^D_{i',c',t'}$ be its dependence. By Equation 4.1, the following constraint must be satisfied

$$M^D_{i',c',t'} + \sum_{t=t_{min}}^{t'+l} M_{i,c,t} \leq 1, \text{ for } t' \in [t_{min}, t_{max}]. \quad (5.2)$$

The constraint requires for each t' that the dependent move must not be executed within time interval $[t_{min}, t' + l]$, where l is latency of the executed operation.

5.2.3 Bypassing and Dead-Result Elimination

Bypass moves generated from the bypass candidate moves are alternative to each other. The bypass move inherits the dependencies from the source move, and each move that is dependent of the bypassed move is dependent of the bypass move as well. Figure 5.1 presents a data dependence graph with an added bypass move, highlighted by the dashed line. The bypass source `ADD.3` $\rightarrow$ `z` and bypass candidate move `z` $\rightarrow$ `SUB.2` are emphasized with the dashed-dotted poly-line. The bypass move contains the incoming edges of the bypass source `ADD.3` $\rightarrow$ `z` and respectively the outgoing edges of the bypass candidate `z` $\rightarrow$ `SUB.2`.

Dead-result elimination allows the removal of the result move in case a move is bypassed to all the destinations. For example, in the situation presented in Figure 5.1 the result move `ADD.3` $\rightarrow$ `z` can be eliminated as the only outgoing edge is to the bypassed move `z` $\rightarrow$ `SUB.2`. In summary, bypassing together with dead-result elimination result in two possible relations of moves. If the result move only has a single outgoing edge, either the bypass move, or the bypass candidate and the result move shall be assigned. However, if the result moves multiple outgoing edges, the result move is to be assigned and the bypass candidate and the bypass move are alternative to each other. By applying Equation 4.4, the former case results in constraint

$$\sum_{c \in C_{\text{candidate}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{candidate},c,t} + \frac{1}{2}\left(\sum_{c \in C_{\text{bypass}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{bypass},c,t} + \sum_{c \in C_{\text{result}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{result},c,t} \right) = 1, \quad (5.3)$$

where $M_{\text{candidate}}$, M_{bypass} and M_{result} are the bypass candidate, the bypass move, and the result move, respectively. The latter condition requires two constraints. The

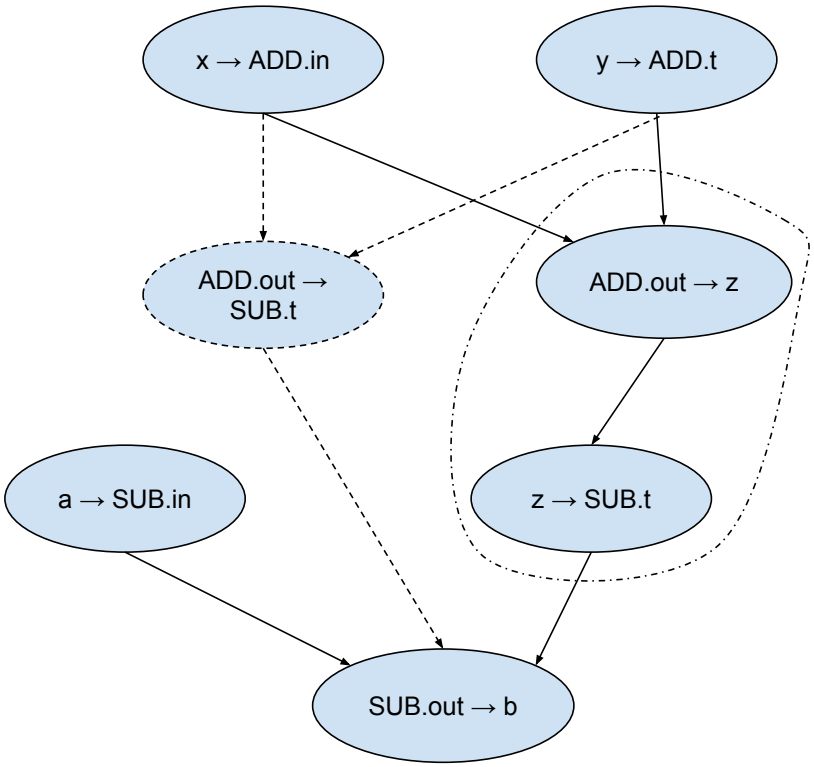

Figure 5.1: Data dependence graph showing a bypass opportunity for move `z` → `SUB.t`

bypass candidate and the bypass move must be alternative to each other. Using Equation 4.4 gives

$$\sum_{c \in C_{\text{candidate}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{candidate},c,t} + \sum_{c \in C_{\text{bypass}}} \sum_{t=t_{min}}^{t_{max}} M_{\text{bypass},c,t} = 1, \tag{5.4}$$

where $M_{\text{candidate}}$ and M_{bypass} are the bypass candidate and the bypass move, respectively. Also the result moves has to be assigned yielding

$$\sum_{c \in C_i} \sum_{t=t_{min}}^{t_{max}} M_{\text{result},c,t} = 1, \tag{5.5}$$

where M_{result} is the result move.

5.2.4 Register File Port Constraints

Register files have a number of registers of which a single one can be read from an output port at any given cycle. In other words, the number of output ports bounds the number of concurrent register reads from a register file. Similarly each input port can write into a single register at a cycle. To limit the outbound moves at each cycle to one for each port we require that

$$\sum_{M_i \in P} \sum_{c \in C_i^{\mathrm{rf}}} M_{i,c,t} = 1, \text{ for } t \in [t_{min}, t_{max}], \mathrm{rf} \in R, \tag{5.6}$$

where R is a set of all RFs on the given TTA processor, and C^{rf} is a set of all connections that originate from register file rf.

5.2.5 Function Unit Constraints

Operations executed at function units consist of multiple operands, of which one is a triggering operand that starts the operation execution. All other operands shall be written to FU input ports at any cycle before or at the same time as the triggering operand. Operand trigger constrains the latest cycle on which other operand must be written. There might be multiple function units that can execute given operation and each operand might have multiple connections to a FU. Therefore by Equation 4.2,

$$\sum_{c \in C^{\mathrm{f}}_{\mathrm{trigger}}} M_{\mathrm{trigger},c,t} - \frac{1}{|O|} \left(\sum_{M_i \in O} \sum_{c \in C_i^{\mathrm{f}}} \sum_{t'=t-e}^{t} M_{i,c,t'} \right) \leq 0,$$
$$\text{for } t \in [t_{min}, t_{max}], \forall \mathrm{f} \in \mathrm{FU}, \tag{5.7}$$

where FU is a set of all function units that can execute the considered operation, C^f is a set of all connections that can transport given move to appropriate port on f, M_{trigger} is the trigger move, O is a set of operand moves that relate to the triggering move M_{trigger}. Variable e is *operand slade limit*, a model parameter that restricts the operand move transport to occur at most $[0, e]$ cycles before the triggering move to reduce model size.

No other operand moves must be written to operand ports between cycles $[t-t', t]$, where t and t' are cycles in which the triggering move and the operand move is being written, respectively. In other words, the operand port is reserved to the operation until the execution triggered to start after the operand is written to the port. This is illustrated in Figure 5.2, where an operation of three operands is being transferred to a FU. The port reservation after an operand write are illustrated with blue line, which spans until the triggering port *FU*.3 is being written into. This yields constraint

$$\sum_{c \in C^{\mathrm{f}}_{\mathrm{trigger}}} M_{\mathrm{trigger},c,t} + \sum_{c \in C^{\mathrm{f}}_{\mathrm{o}}} M_{\mathrm{o},c,t'} + \frac{1}{|K|} \left(\sum_{M_i \in K} \sum_{c \in C^{\mathrm{f}}_i} \sum_{t''=t'}^{t} M_{i,c,t'} \right) \leq 2,$$
$$\text{for } t \in [t_{min}, t_{max}], \forall t' \in [t-e, t], \forall \mathrm{f} \in \mathrm{FU}, \forall M_o \in O, \quad (5.8)$$

where M_o is the current operand node, O is the set of all operand moves, and K is a set of other operand moves that might be assigned to same port as M_o. The constraint is an application of Equation 4.4. By setting the left-hand side be equal to two both the triggering move and the current operand move can be equal to one, and all the other operand moves must be zero.

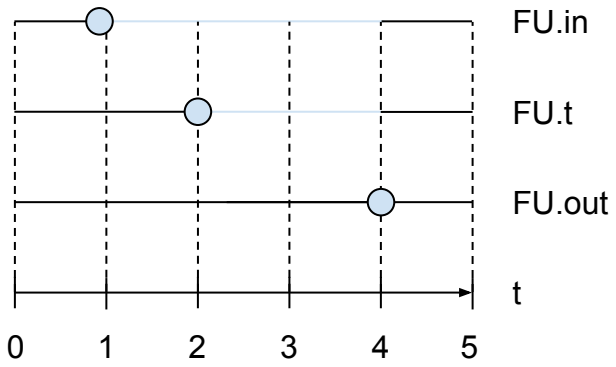

Figure 5.2: Function unit port reservations until an operand is written. The first two operands, `FU.in` and `FU.t`, reserve the corresponding input ports until the execution is triggered to start from port `FU.out`.

The results are available at the output ports after an operation latency has passed since the triggering move has been written and until the next operation overwrites the result. All result reads of an operation must be read some time after the triggering move has been written, and no other operation can overwrite the result before the result read occurs. Figure 5.3 illustrates the port reservation in case of two operations in which `FU.2` is the triggering port and both have a latency of two cycles. The first operation result is available at $t = 4$, that is operation latency after the trigger `FU.2` is written. Since the result of the first operation is read at $t = 5$, the execution of operation two shall not start before $t = 4$.

Using Equation 4.3 to enforce all result moves to occur at correct interval leads to

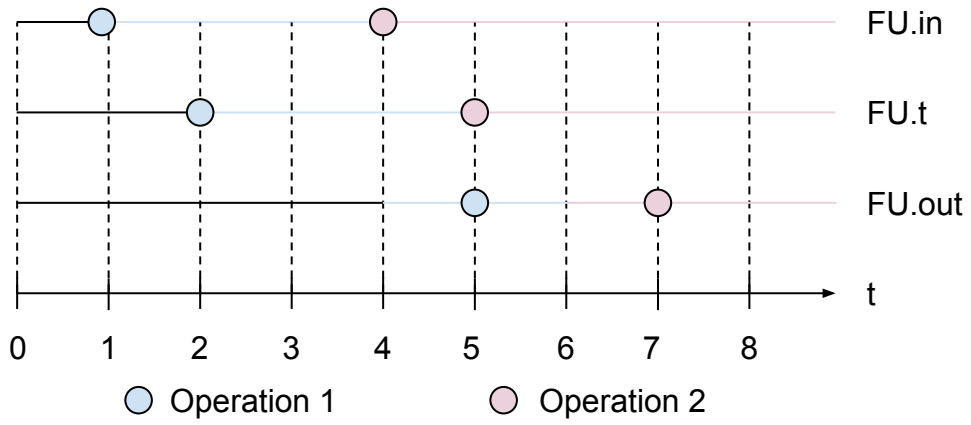

Figure 5.3: Pipelining of two operations. The first operation trigger cycle determines when the results are available. Similarly, the cycle of the last result read determines when the execution of next operation can be started.

$$\sum_{c \in C^{\mathrm{f}}_{\mathrm{trigger}}} M_{\mathrm{trigger},c,t} - \frac{1}{|R|} \sum_{M_i \in R} \sum_{c \in C^{\mathrm{f}}_i} \sum_{t'=t+l}^{t+l+e} M_{i,c,t'} \leq 0,$$

$$\text{for } t \in [t_{min}, t_{max}], \forall \mathrm{f} \in \mathrm{FU}, \quad (5.9)$$

where R is a set of all results reads of the operation that M_{trigger} triggers, and e is result read slack, a maximum cycle count that given result can be retained in an output port. All other triggering moves must be prevented to overwrite the result

$$\sum_{c \in C^{\mathrm{f}}_{\mathrm{trigger}}} M_{\mathrm{trigger},c,t} + \sum_{c \in C^{\mathrm{f}}_{\mathrm{r}}} M_{\mathrm{r},c,t'} + \frac{1}{|T|} \left(\sum_{M_i \in T} \sum_{c \in C^{\mathrm{f}}_i} \sum_{t''=t'}^{t} M_{i,c,t'} \right) \leq 2,$$

$$\forall t \in [t_{min}, t_{max}], \text{ for } t' \in [t-e, t], \forall \mathrm{f} \in \mathrm{FU}, \forall M_r \in R, \quad (5.10)$$

where T is a set of other trigger moves to function unit f besides M_{trigger}, and R is a set of result moves of M_{trigger}. The reasoning behind the constraint is identical to that of Equation 5.8.

5.3 Special Ordered Sets

This section provides special ordered sets for the model described in the previous section. These sets may be used by branch and bound solvers The ordering of the special ordered sets defined below is based on the variable cycle, the earlier a variable is assigned the better. This prioritizes as early assignments of variables as possible.

5.3.1 SOS1: All Moves Are Assigned Once

Since all must moves must be assigned only once as per Equation 5.1, variables generated for each move of a program are mutually exclusive to each other. This yields the following SOS1

$$\sum_{c \in C_i} \sum_{t=t_{min}}^{t_{max}} M_{i,c,t} \leq 1. \tag{5.11}$$

5.3.2 SOS1: Input Socket Constraints

An input socket is able to transfer a single move into the unit on each cycle:

$$\sum_{M_i \in P} \sum_{c \in C^s} \sum_{t=t_{min}}^{t_{max}} M_{i,c,t} \leq 1, \text{ for } s \in I, \tag{5.12}$$

where I is a set of all input sockets and C^s is a set of all connections that pass along socket s.

5.3.3 SOS1: Bus Constraints

A bus is capable of transferring a single value between two connected sockets on each cycle. This yields SOS for all buses on each cycle

$$\sum_{M_i \in P} \sum_{c \in C^b} \sum_{t=t_{min}}^{t_{max}} M_{i,c,t} \leq 1, \text{ for } b \in B, \tag{5.13}$$

where B is a set of all buses and C^b is a set of connections that consume bus b.

5.4 Objective Function

Objective function of the integer linear programming model can be varied depending on the intended use of the designed TTA processor. If the most important criterion is the execution time of the compiled program, the height of the DDG can be minimized. To achieve that, the objective function attempts to execute the leaf nodes of the data dependence graph as early as possible. Thus all the other nodes of the graph are pushed to be assigned earlier as they all are descendants of the leaf nodes. The objective function to minimize is

$$\min : \sum_{M_i \in L} \sum_{c \in C^b} \sum_{t=t_{min}}^{t_{max}} t \cdot M_{i,c,t}, \tag{5.14}$$

where L is a set of leaf nodes of the data dependence graph. The model must minimize the cycle of all leaf nodes and not just the ones on the critical path because

otherwise there might be outlier leaf nodes that increase the cycle count of the basic block.

6. EMPIRICAL EVALUATION

The integer linear programming model for the TTA scheduling problem was tested on a server that had processor clock speed of 3.2GHz, 12 CPU cores, and 16384MB random-access memory. The models were optimized with Solving Constraint Integer Programs (SCIP) Mixed Integer Programming solver, which is one of the fastest non-commercial solvers available [30]. The model was compared to the graph-based heuristic scheduler that TCE uses by default [8].

For each input program compiled, the cycle count, and the number of register accesses were registered. The cycle count and the register usage of the compiled program were obtained from the TTA simulator described in Subsection 2.3.3. In addition, the CPU time and the wall clock time of the TCE compiler were measured with the GNU time utility (version 1.7), which is accurate enough for these kinds of measurements.

The sections below present the results for two different transport triggered architecture processors. The objective function used is the one described in Subsection 5.4, in which the height of the data dependence graph is minimized. There were five different example problems: infinite impulse response (IIR) filter, finite impulse response (FIR) filter, dot product calculation, convolution operation, and complex number arithmetics. These kinds of problems are typically executed in embedded processors and contain a reasonable amount of concurrency.

6.1 Minimalist Architecture

Minimalist architecture consists of two register files, RF and BOOL, an ALU, a LSU, a CU, and a function unit MUL providing the multiplication operation. There are three buses in the architecture. The interconnection network is quite reduced, but provides enough connections to execute multiple operations concurrently. The ALU can transport results back to its input ports enabling bypassing between arithmetic operations. In addition, it is possible to bypass results from the MUL unit to the ALU, and the other way. The load-store unit results and inputs can be bypassed directly from both the ALU unit and the MUL unit. The architecture is shown in Figure 6.1.

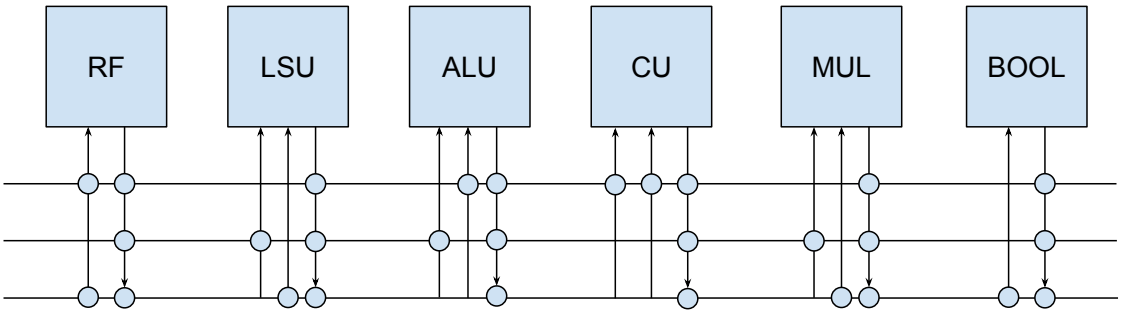

Figure 6.1: Minimalist architecture with two register files, RF and BOOL, an ALU, a LSU, a CU, and a function unit MUL providing the multiplication operation.

The results for the minimalist architecture are tabulated in Appendix 1. The integer linear programming scheduler is able to reduce the input program into 52% of the cycles compared to the heuristic scheduler in the best case, and 97.5% at worst case. The average cycle count equals 70.0% of the cycle count of the heuristic scheduler. Relative cycle counts are shown in Figure 6.2.

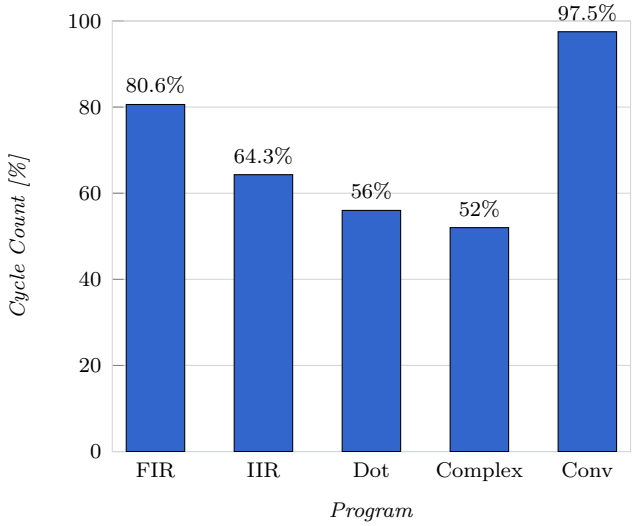

Figure 6.2: Integer linear programming scheduler cycle count relative to that of the heuristic scheduler. The percentages are obtained by dividing the cycle count of the integer linear programming scheduler by the cycle count of the heuristic scheduler.

The reduction of register reads is apparent although the objective was to minimize the cycle count. In some cases, the IIR filter and the dot product calculation, the integer linear programming scheduler was able to eliminate all results reads. The schedule produced for the convolution program had slightly more register reads than the heuristic scheduler. On average, the integer linear programming scheduler consumed 64.7% less register reads compared to the schedule of the heuristic scheduler. The integer linear programming scheduler was able to eliminate 25.1% of the

register writes on average. In the case of the convolution program, the amount of register reads was equal to that of the heuristic scheduler. The register accesses of the integer linear programming scheduler are presented in Figure 6.3.

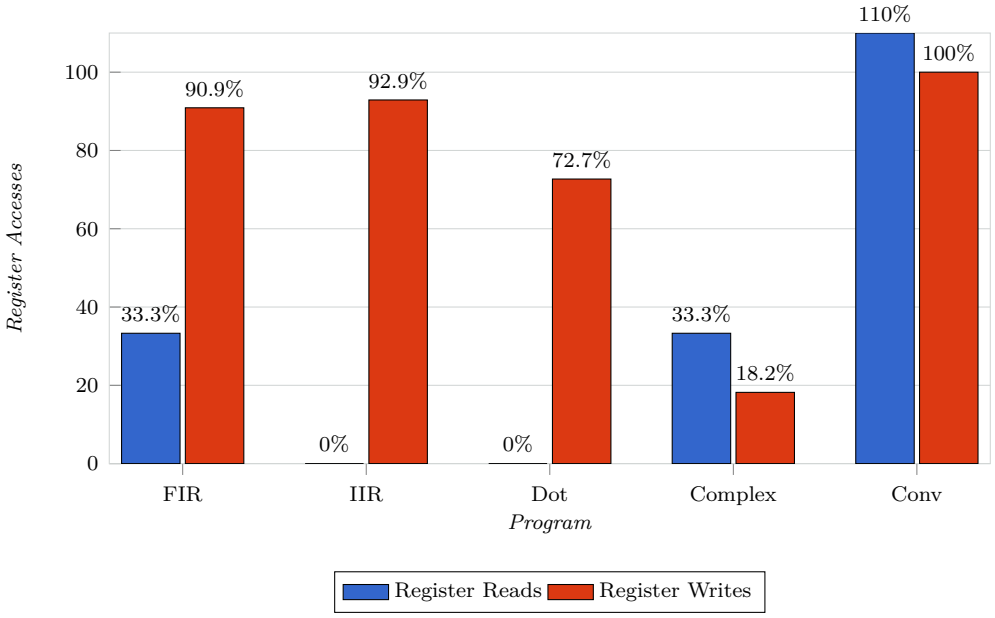

Figure 6.3: The ratio of register reads and register writes between the integer linear programming scheduler and the heuristic scheduler.

Figure 6.4 presents the execution times for the integer linear programming scheduler. The heuristic scheduler was able to schedule all programs in a second or less. It is evident that the proposed scheduler takes distinctly more time to schedule the programs, 4.7 minutes on average and 19.1 minutes in the worst case.

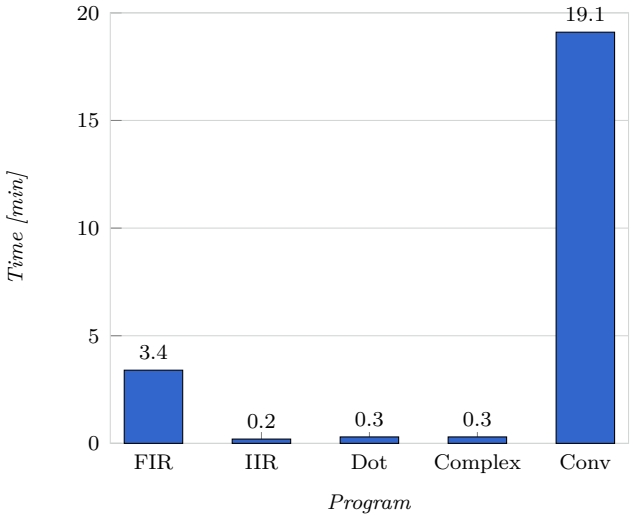

Figure 6.4: Execution times for the integer linear programming instruction scheduler.

6.2 Clustered Architecture

Clustered architecture is a VLIW-like architecture divided into separate computing clusters. These three clusters are formed of a register file and an ALU pairs. The clusters are interconnected into each other. In addition, the architecture provides a MUL unit for multiplication operation. There are 17 buses altogether. Although the interconnection network is quite reduced, the architecture provides a high level of concurrency for the operations. The architecture is illustrated in Figure 6.5.

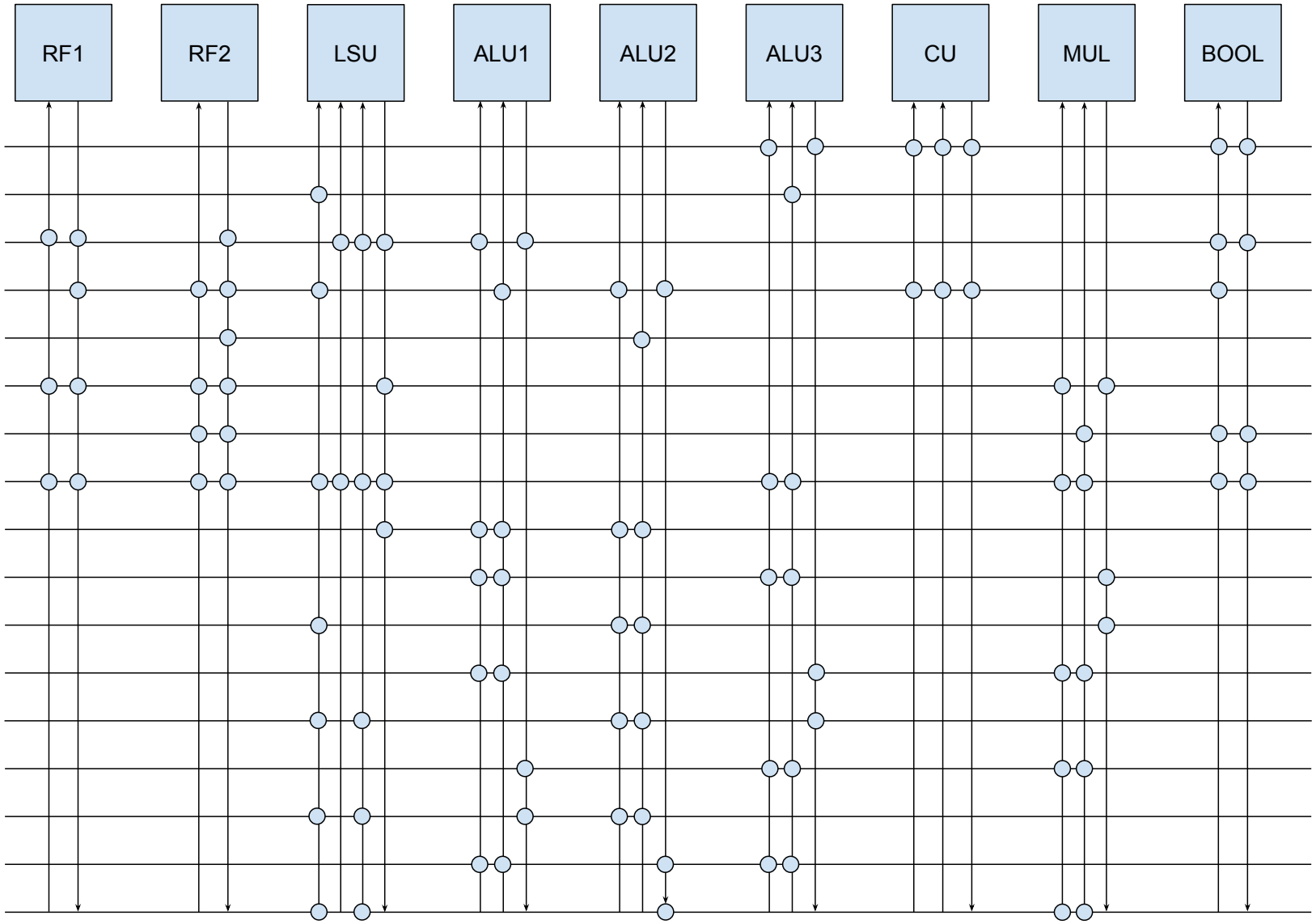

Figure 6.5: Clusted architecture consisting of three separate computing clusters, a multiplication unit, a CU, a LSU, and a boolean register file.

The results for the clustered architecture are tabulated in Appendix 1. On average, the integer linear programming scheduler decreased the program length by 19.6%. The most notable reduction was in the case of the complex number arithmetics program, in which the number of cycles were 43.7% less than that of the heuristic scheduler. The cycle counts of the integer linear programming scheduler are presented in Figure 6.6.

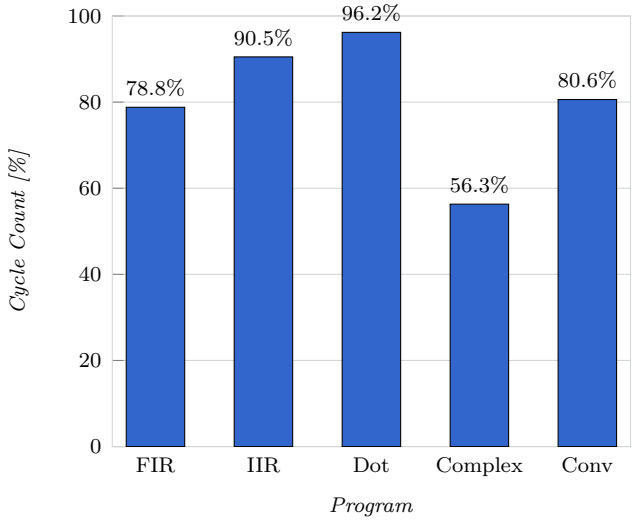

Figure 6.6: Integer linear programming scheduler cycle count relative to that of the heuristic scheduler. The percentages are obtained by dividing the cycle count of the integer linear programming scheduler by the cycle count of the heuristic scheduler.

The schedules produced with the integer linear programming scheduler executed 45.7% less register reads compared to the heuristic scheduler. In the case of the IIR filter, the integer linear programming scheduler was able to eliminate all register reads. On the other hand, there were 12.2% less register writes as to the heuristic scheduler. Register accesses of the integer linear programming scheduler with the clustered architecture are shown in Figure 6.7.

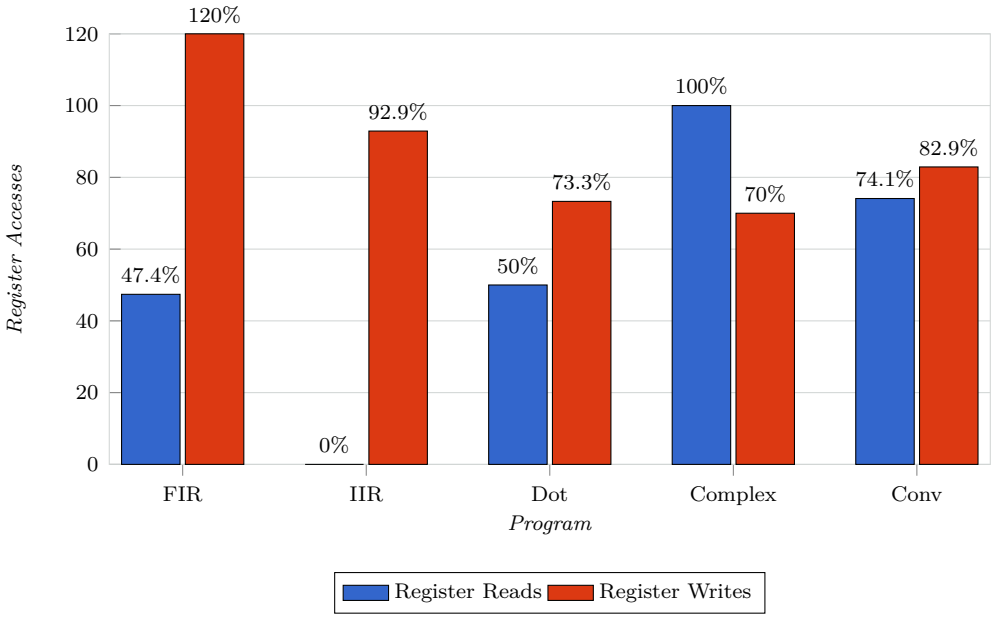

Figure 6.7: The ratio of register writes and register reads between the integer linear programming scheduler and the heuristic scheduler.

Figure 6.8 shows the execution times for the integer linear programming scheduler. The execution times are even greater than those of the minimalist architecture. This is due to the fact that the clustered architecture is much larger and provides much more opportunities for data transports between the units. The average execution time for the clustered architecture equals 7.2 minutes. In worst case, the proposed scheduler uses 21 minutes to produce a schedule for the FIR filter program.

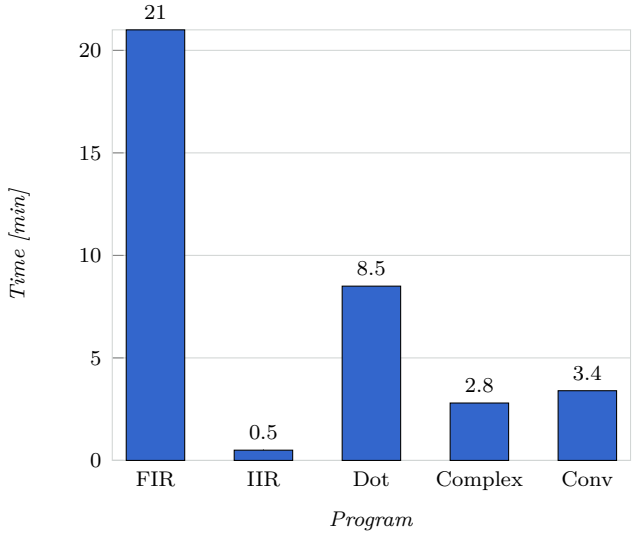

Figure 6.8: Execution times for the integer linear programming instruction scheduler.

6.3 Discussion

The objective of this thesis work was to model the transport triggered architecture instruction scheduling problem as an integer linear programming model. In addition, this model was to be optimized for the cycle count of the input program. With both architectures, the minimalist architecture and the clustered architecture, the integer linear programming based scheduler was able to produce more concise programs in terms of the cycle count compared to the heuristic scheduler of the TCE compiler. On average, the schedules of the integer linear programming model were 24.7% shorter. In the most notable case, the schedule had 48% less cycles than that of the heuristic scheduler. The reduction of cycles is achieved by utilizing the parallel resources of the given architecture further. The heuristic scheduler has a limited number of schemes to exploit on scheduling, and it works on sequential manner. That is, each decision it makes might worsen the outcome of the following decisions, but it does not backtrack in those cases. Whereas the integer linear programming scheduler explores much wider result space and is able to find improved results.

The cycle count reduction was notably higher in the case of the minimalist architecture. The architecture has less possibilities for concurrent transports, and thus is harder to schedule for. In contrast, the clustered architecture has much more parallel connections which makes the scheduling easier. This leads to the conclusion that the harder an architecture is for a scheduling algorithm, the better the integer linear programming scheduler performs when compared to the heuristic scheduler.

In addition to the cycle count, the amount of register accesses of the programs were registered. A register read can be eliminated by bypassing the value directly from the origin unit to the target unit. Whereas a register write can be eliminated through dead-result elimination. That is, if all uses of a value in a register are bypassed, then the result move into the register can be removed. While the decrease in register access was not the optimization objective, it is tightly connected to the reduction of the cycle count. This is due to the fact that bypassing and dead-result elimination reduce the amount of register access, and simultaneously eliminates intermediate result transports through registers, which in turn allows more rapid execution of operations.

The integer linear programming scheduler was able to eliminate 55.2% of the register reads compared to the heuristic scheduler on average. Correspondingly, 18.6% of the register writes were eliminated from the schedules. In two cases, the integer linear programming scheduler resulted in excess register access when compared to the heuristic scheduler, but the schedules were nonetheless shorter so the objective was met.

The execution time of the integer linear programming scheduler was an order of magnitude greater in all cases compared to the heuristic scheduler. The average execution time was 4.7 minutes for the minimalist architecture, and 7.2 minutes for the clustered architecture. In contrast, the heuristic scheduler was able to schedule all programs in a second or less. This difference in execution time was expected as the integer linear programming solvers attempts to find the optimal solution given the objective.

The execution time of a program depends on the architecture used, the complexity of the data dependence graph, the amount of moves in the program, and the model parameters. It is difficult to predict the execution time in advance by looking at the high-level language code and the defined architecture as the number of influential parameters is high and their mutual relations affect the outcome as well.

7. CONCLUSIONS

An efficient scheduling algorithm for transport triggered architecture processor is important because it might lead to significant improvements in terms of chip area, energy consumption, and clock frequency. This thesis presented an integer linear programming based scheduling model for the transport triggered processor architecture. The model was implemented to the TCE retargetable compiler backend as a scheduling pass, and can be used to schedule TTA programs. The scheduler attempts to produce optimal schedules by some given criterion, for example to minimize the energy consumption.

The empirical evaluation was based on two different transport triggered architectures, the minimalist architecture and the clustered architecture. The former is a simple architecture containing only the essential components, and the latter is a VLIW-style clustered architecture providing a high level of concurrency. The optimization objective was to minimize the cycle count of the schedule. The integer linear programming scheduler was able to produce on average 24.7% shorter schedules for both architectures compared to the heuristic scheduler of the TCE compiler.

In addition to the cycle count, the amount of register accesses was measured. Even though the register usage was not the optimized parameter, it is closely related to the reduction of the cycle count. Generally the less register access is utilized, the shorter the schedule. This is due to the fact that bypassing and dead-result elimination decreases the register usage and frees register file ports to other moves, and at the same time eliminates intermediate result transports through registers, which in turn causes the schedule to be shorter. The integer linear programming based scheduler was able to eliminate 55.2% of the register reads, and 18.6% of the register writes on average. In two cases register accesses in the schedules produced by the proposed scheduler exceeded those of the heuristic scheduler, but the schedules were nonetheless shorter.

The execution times of the integer linear programming based scheduler were distinctly longer than those of the heuristic scheduler. This was expected as the integer linear programming solver explores much wider solution space than the heuristic scheduler. The average execution times was 6 minutes, and 21 minutes at the worst case. That being said, we argue that the execution time in this context is somewhat

unimportant, as the designed architecture is possibly used to produce a large number of processors. A small cost saving accumulates into significant amount when enough hardware is fabricated. Consequently, it is acceptable to use more execution time if it leads to better design.

7.1 Future work

This thesis presented an integer linear programming model for the transport triggered architecture scheduling problem. The model covers the basic properties of the architecture, but there is much room for further development. The TCE infrastructure contains numerous optimizations that are done to make the compiled programs more efficient. Many of these improvements were not integrated to work with the integer linear programming scheduler at this time. In addition, there are a few additional things that the model could include as well to be able to make more intelligent decisions.

Currently the register allocation in done prior to scheduling for reasons stated at Section 3.2.4. This distinctly restricts the scheduler's freedom to make decisions and thus leads to inferior schedules as some of the move sources are determined in advance. In order for the current integer linear programming model to also assign registers to variables, it would need to include information about each variable and the memory location in which the variable exists at any given cycle.

Function units might have internal shared resources that limit the pipelining of operations in the FU. Currently the model assumes that all operations are fully independent and permits pipelining for as long as operations do not overwrite others' results. To cope with shared resources, the internal resources should be modeled and limited at any given cycle to be used by at most a single operation.

Immediate values are coded to the source port field of the TTA move operation. This means that the operation source fields needs to be as wide as the largest supported constant value. In practice, TCE templates allow buses have different immediate widths and larger values need to be transfered from *Immediate Unit* (IU), which is a special unit to store constant values. Different immediate widths and the immediate unit are not currently supported in the model.

Instruction selection is currently performed before the scheduling which limits the ability to achieve optimal schedule. Some instruction templates might lead to more desirable end result even though they might be inferior from instruction selection perspective. The integer linear programming model should be changed so that there are multiple mutually exclusive sets of moves, the different instruction selections, of which only a single set should be assigned.

Modulo scheduling is a form of software pipelining in which the iterations of a loop use the same schedule [31]. These iterations are initiated at a constant rate.

Initiation Interval (II) is a constant that bounds the number of cycles between two consecutive iterations of the loop. II depends on the loop operation dependencies and the processor resources, and can be found using iterative search. Extending the integer linear programming scheduling model to support modulo scheduling would lead to higher performance and more compact code for loop kernels.

Further research may also concern alternative objective functions to achieve different goals. For example, minimizing the register accesses of the TTA program has significant effect on the energy consumption albeit the compiled program could be longer in terms of cycles. This could be achieved with added cost for each register read and write. Secondary optimization criterion usually still needs to be the schedule length due to real time constraints.

The current implementation of the model in TCE is not able to handle large basic blocks. This is limitation of both the model, especially the function unit constraints, and the implementation. The model places unnecessary and sometimes overlapping constraints that could be reduced with careful review. For example, function unit constraints could be pruned using the data dependence graph since some operations can not happen after or before others. Further, the current TCE scheduling algorithm could be used to provide an initial suboptimal solution for the integer linear programming solver to speed up the solving. In addition, a preliminary schedule could be used to set the model cycle count, that is parameters $t_{\min}$ and $t_{\max}$, more intelligently.

Presenting the constraints as $0-1$ integer linear programming model sometimes leads to quite large models in terms of size due to a large number of variables, which in turns leads to a high memory usage on solving. Using ordinary integer variables might be worth the try to see how much the memory consumption and model size are decreased. Related to this, it would also be useful to only optimize the most important parts of the program with the integer linear programming scheduler and use the suboptimal but much faster heuristic scheduling algorithm for the rest.

REFERENCES

[1] H. Corporaal, J. Janssen, and M. Arnold, "Computation in the context of transport triggered architectures," *International Journal of Parallel Programming*, vol. 28, no. 4, pp. 401–427, 2000.

[2] D. She, Y. He, B. Mesman, and H. Corporaal, "Scheduling for register file energy minimization in explicit datapath architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '12. San Jose, CA, USA: EDA Consortium, 2012, pp. 388–393.

[3] B. Rister, P. Jaaskelainen, O. Silven, J. Hannuksela, and J. Cavallaro, "Parallel programming of a symmetric transport-triggered architecture with applications in flexible LDPC encoding," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, May 2014, pp. 8380–8384.

[4] H. Torng and S. Vassiliadis, *Instruction-Level Parallel Processors*. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[5] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *ISCA '83: Proceedings of 10th International Symp. on Computer Architecture*, Los Alamitos, CA, Jun. 1983, pp. 140–150.

[6] J. Hoogerbrugge and H. Corporaal, "Register file port requirements of transport triggered architectures," in *Microarchitecture, 1994. MICRO-27. Proceedings of the 27th Annual International Symposium on Computer Architecture*, Nov 1994, pp. 191–195.

[7] H. Corporaal, "Transport triggered architectures, design and evaluation," Ph.D. dissertation, Delft Univ. Tech., Netherlands, 1995.

[8] O. Esko, P. Jääskeläinen, P. Huerta, C. de La Lama, J.Takala, and J. Martinez, "Customized exposed datapath soft-core design flow with compiler support," in *Proceedings of International Conference Field Programmable Logic and Applications*, Milan, Italy, Aug. 2010, pp. 217–222.

[9] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[11] G. Hjort Blindell, "Survey on instruction selection : An extensive and modern literature review," KTH, Software and Computer systems, SCS, Tech. Rep. 13:17, 2013, qc 20131007.

[12] G. J. Chaitin, "Register allocation & spilling via graph coloring," *SIGPLAN Not.*, vol. 17, no. 6, pp. 98–101, Jun. 1982.

[13] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation: What does the NP-Completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, vol. 4382, pp. 283–298.

[14] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 36–44, Jun. 1985.

[15] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation Optimization*, Mar. 2004, pp. 75–87.

[16] F. Rossi, P. v. Beek, and T. Walsh, *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. New York, NY, USA: Elsevier Science Inc., 2006.

[17] S. Bradley, A. Hax, and T. Magnanti, *Applied mathematical programming*. Addison-Wesley Pub. Co., 1977.

[18] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, ser. STOC '71. New York, NY, USA: ACM, 1971, pp. 151–158.

[19] L. Fortnow, "The status of the P versus NP problem," *Communications of the ACM*, vol. 52, no. 9, pp. 78–86.

[20] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.

[21] H. T. Jongen, K. Meer, and E. Triesch, *Optimization theory*. Kluwer, 2004.

[22] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, ser. The IBM Research Symposia Series, R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, Eds. Springer US, 1972, pp. 85–103.

[23] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[24] C. H. Papadimitriou, "The Euclidean travelling salesman problem is NP-complete," *Theoretical Computer Science*, vol. 4, no. 3, pp. 237–244, 1977.

[25] G. Reinelt, *The Traveling Salesman: Computational Solutions for TSP Applications*. Berlin, Heidelberg: Springer-Verlag, 1994.

[26] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton, NJ, USA: Princeton University Press, 2007.

[27] J. Clausen. Branch and bound algorithms - principles and examples. (1999) [Accessed on 16th of Sep. 2014]. [Online]. Available: http://janders.eecg.toronto.edu/1387/readings/b_and_b.pdf

[28] A. Shmuel, "The relaxation method for linear inequalities," *Canadian Journal of Mathematics*, no. 6, pp. 382–392, 1954.

[29] E. Beale and J. Tomlin, "Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables," in *Proceedings of the Fifth International Conference on Operational Research*. London: Tavistock Publications, 1970, pp. 447–454.

[30] T. Achterberg, T. Berthold, T. Koch, and K. Wolter, "Constraint integer programming: A new approach to integrate CP and MIP," in *Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, ser. CPAIOR'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 6–20.

[31] M. Lam, "Software pipelining: An effective scheduling technique for vliw machines," *SIGPLAN Not.*, vol. 23, no. 7, pp. 318–328, Jun. 1988.

APPENDIX 1: COMPLETE RESULTS

This appendix presents the complete results for the two architectures used in the empirical evaluation. For each input program compiled, the cycle count, and the number of register accesses were registered. The cycle count and the register usage of the compiled program were obtained from the TTA simulator. In addition, the CPU time and the clock time of the TCE compiler were measured.

Table 1 displays the results for the minimal architecture for the five benchmark programs.

	Integer Linear Programming Scheduler						**Heuristic Scheduler**			
Program	**Cycle Count**		**Register Reads**		**Register Writes**		**Compilation Time [min]**	**Cycle Count**	**Register Reads**	**Register Writes**	**Compilation Time [s]**
FIR filter	25	(80.6%)	5	(33.3%)	10	(90.9%)	3.4	31	15	11	1
IIR filter	18	(64.3%)	0	(0.0%)	13	(92.9%)	0.2	28	9	14	1
Dot product	14	(56.0%)	0	(0.0%)	8	(72.7%)	0.3	25	3	11	1
Complex	13	(52.0%)	3	(33.3%)	2	(18.2%)	0.3	25	9	11	1
Convolution	78	(97.5%)	33	(110%)	35	(100.0%)	19.1	80	30	35	1

Table 1: Simulation results for the minimal architecture with the integer linear programming scheduler and the heuristic scheduler for the benchmark programs. In the integer linear programming columns, the percentage in the parentheses shows the difference with respect to the corresponding value of the heuristic scheduler.

Table 2 shows the results for the clustered architecture for the five benchmark programs.

	Integer Linear Programming Scheduler							Heuristic Scheduler			
Program	**Cycle Count**		**Register Reads**		**Register Writes**		**Compilation Time [min]**	**Cycle Count**	**Register Reads**	**Register Writes**	**Compilation Time [s]**
FIR filter	26	(78.8%)	9	(47.4%)	12	(120.0%)	21.0	33	19	10	1
IIR filter	19	(90.5%)	0	(0.0%)	13	(92.9%)	0.5	21	6	14	1
Dot product	25	(96.2%)	7	(50.0%)	11	(73.3%)	8.5	26	14	15	1
Complex	9	(56.3%)	5	(100.0%)	7	(70.0%)	2.8	16	5	10	1
Convolution	29	(80.6%)	20	(74.1%)	29	(82.9%)	3.4	36	27	35	1

Table 2: Simulation results for the clustered architecture with the integer linear programming scheduler and the heuristic scheduler for the benchmark programs. In the integer linear programming columns, the percentage in the parentheses shows the difference with respect to the corresponding value of the heuristic scheduler.